

Community Experience Distilled

GameMaker Game Programming with GML

Learn GameMaker Language programming concepts and script integration with GameMaker: Studio through hands-on, playable examples

Matthew DeLucas

[PACKT] PUBLISHING

Table of Contents

GameMaker Game Programming with GML
Credits
About the Author
About the Reviewers
www.PacktPub.com
    Support files, eBooks, discount offers and more
        Why Subscribe?
        Free Access for Packt account holders
Preface
    What this book covers
    What you need for this book
    Who this book is for
    Conventions
    Reader feedback
    Customer support
        Downloading the example code
        Downloading the color images of this book
        Errata
        Piracy
        Questions
1. Getting Started – An Introduction to GML
    Creating GML scripts
        Creating GML scripts within an event
        Creating scripts as resources
        Scripting a room's creation code
    Understanding parts of GML scripts
        Programs
            snake_case
        Variables
            Variable prefixes
            Variable scope
                Built-in variables
                Creating custom constants

- Functions and accessing script resources
  - Arguments
- Expressions
  - Expression symbols
- Conditional statements
  - if, else, and switch
  - repeat, while, do, and for
  - break, continue, and return
- Arrays
  - Two-dimensional arrays
- Commenting
- Errors

Pushing your buttons
- Creating the project
- Gathering resources
  - Sprites – spr_button
  - Objects – obj_button
  - Room – rm_main
- The events of obj_button
  - The Create event
  - The Left Button event
  - The Left Pressed event
  - The Mouse Enter event
  - The Mouse Leave event
  - The Global left release event
- The Draw GUI event
  - Scripts – scr_create_button
  - scr_random_position
- Creating buttons using scripts
  - Scripting room creation code
  - Creating scr_random_position
- Exporting and importing the button

Summary

2. Random Organization – Creating a Puzzle Game
- Understanding sprite variables and functions
  - Positioning and origin
    - Layering with depth

Rotating
Scaling
Working with frames and color
Setting up the puzzle game
Sprites – spr_grid
spr_pieces
Objects – obj_grid_manager
obj_grid_block and obj_puzzle_piece
Room – rm_gameplay
Scripts
Aligning the pieces of the puzzle game to a grid
The Create event of obj_grid_manager
scr_create_in_grid
Understanding and utilizing randomization
Random functions
Randomizing the board
Checking pieces
scr_get_puzzle_piece
scr_check_adjacent
scr_check_board
Running it all together
Summary
3. So How Do I Play? – Adding Player Interaction
Designing player interaction
Reading input with the mouse
Understanding mouse variables and functions
Working with touch devices
Creating resources to integrate mouse input into the puzzle game
Adding new events to obj_grid_manager
The Create event
Updating spr_grid and obj_grid_block
Adding new frames to spr_grid
Implementing the Mouse events for obj_grid_block
Entering the grid with the mouse
Pressing the left mouse button
Releasing the left mouse button globally
The Mouse Leave event

- Swapping pieces
- obj_puzzle_piece
  - scr_swap_pieces
- Updating scr_check_board
- Updating organization with scr_reorganize_board
- Integrating keyboard input
- Introducing keyboard functions, variables, and events
- Integrating the Keyboard event updates
  - Utilizing the press <any key> event
  - Implementing the release <any key> event
- Summary

4. Juicy Feedback – Aural and Visual Effects
- Introducing alarms
- Arming the puzzle game's alarms
- Setting up the first alarm
- Are pieces shifting?
  - Applying is_shifting_pieces
- Determining where pieces should move
  - Setting the Alarm 0 event of obj_puzzle_piece
  - Making pieces fall with Alarm 1 in obj_puzzle_piece
- Setting the alarms
  - Updating scr_swap_pieces
  - Updating scr_check_board
  - Updating scr_reorganize_board
- Hiding the pin drop – adding audio
- Creating sound resources
- Gathering audio resources
- Introducing audio functions
- Playing puzzling sounds
  - Swapping sounds in scr_swap_pieces
  - Playing the pop sound in scr_reorganize_board
  - Fading in the music with obj_grid_manager
- Visualizing effects with particles
- Managing a particle system
- Emitting particles
  - Shaping and distributing within emitters
- Creating different particle types

Shaping particle types
Particle-specific functions
Integrating particles
obj_particle_manager
The Create event
The Destroy event
Placing obj_particle_manager
Creating bursts within scr_reorganize_board
Summary
5. Solving the Puzzle – Finishing Touches to the Puzzle Game
Drawing and graphical user interface
Understanding draw functions
Drawing rectangles
Setting color and alpha
Drawing circles and ellipses
Setting a circle's precision
Drawing points, lines, and arrows
Drawing text
Setting font and text alignment
Drawing sprites
Tiling sprites
Establishing the drawing order
Gathering resources for creating the main menu
Creating obj_main_menu
Building a new room – rm_main_menu
Creating fonts – fnt_title and fnt_main
Creating the fonts
Scripting obj_main_menu
The Create event
Utilizing div and mod
Drawing the menu
The Global Left Button event
The press <any key> event
The Global Left Release event
The release <any key> event
event_perform
scr_on_menu_button_pressed

- Changing obj_grid_manager
- Integrating score and time
  - Scoring and combos
  - The Create event of obj_grid_manager
  - Earning points in scr_reorganize_board
  - Drawing UI in obj_grid_manager
  - Timing and game over
  - Adding new variables to the Create event of obj_grid_manager
  - Using Alarm 1 and Alarm 2 in obj_grid_manager
    - Alarm 1
    - Alarm 2
  - Drawing the timer
- Summary

6. Finite State Machines – Starting the 2D Platformer
- Introducing finite state machines
- Gathering resources for the platformer
  - Establishing Lil' Vlad's sprites
    - The standing sprite – spr_vlad_idle
    - The walking sprite – spr_vlad_walk
    - The jumping sprite – spr_vlad_jump
  - Jumping with sound – snd_jump
  - Creating a new object resource – obj_vlad
    - Utilizing the User defined events
  - Placing Vlad in a room – rm_level1
- Defining Vlad's state constants
- Starting Vlad's events and scripts – walking
  - The Create event
  - The Step event
  - Standing still – the User Defined 0 event
  - Walk this way – the User Defined 1 event
  - Adding new variables for jumping
  - Using up to jump – the Step event update
  - Falling state – the User Defined 2 event
  - Looping the jump – the Animation End event
- Summary

7. It's in the Name – Platforms and Collisions
- Collision – a crash course

Creating masks for collision
Working with placement and movement functions
Testing placement
Movement functions
Gathering resources to build platforms
Sprites – spr_basic_platform and spr_solid_platform
Objects – obj_basic_platform and obj_solid_platform
Solidifying objects
Populating the room
Working with Collision events
Updating the Create event of obj_vlad
The Collision events of obj_vlad
Creating the script resource scr_test_collision
Updating the Step event
Moving platforms with paths
Creating path resources
Utilizing the path_start function
Gathering resources for the path creation
The spr_moving_platform sprite
obj_moving_platform
pth_horizontal and pth_vertical_ellipse
Integrating the moving platforms
Creating instances of obj_moving_platform
Interacting with obj_moving_platform
Creating a new variable – current_platform
Updating the User Defined 2 event
scr_test_collision and current_platform
Using the Step End event
Drawing a path
Preventing Vlad from leaving
Defining global.room_left and global.room_right
Updating the End Step event
Knowing the design ahead of time (when possible)
Summary
8. Setting the Stage – Views, Backgrounds, and Tiles
Expanding the room – views
Setting the view

- Adjusting view parameters
- Preparing the game for obj_camera
  - The Create event
  - The End Step event
  - Adding the camera with Creation Code
- Setting the environment – backgrounds
  - Introducing background variables
  - Creating background resources
  - Preparing the game for background resources
    - Building an atmosphere with bg_main
  - Utilizing the background
  - Setting a background index with bg_index
  - Scripting scr_inverse_lerp
    - Moving the background in the End Step event of obj_camera
- Introducing tiles
  - Creating tiles
  - Building resources for tiles
    - Tiling with bg_tiles_main
  - Applying tiles
  - Utilizing tile_add and other tile functions
  - Placing tiles with scripts – scr_define_tiles
  - Using scr_define_tiles
  - Drawing tiles in obj_moving_platform
- Summary

9. Breaking Vlad – Pickups, Hazards, and Enemies
- Tracking health with Draw and Draw GUI
  - Displaying UI with the Draw and Draw GUI events
    - Creating a new font – fnt_score_text
    - Setting up the Draw GUI event
    - Displaying health with the Draw event
    - Updating the Create event
    - Arming the Alarm 0 event in obj_vlad
- Working with pickups
  - Gathering resources to create pickups
    - Initializing sprite resources for pickups
    - Pickup object resources
    - Script resource – scr_collect_pickup

Colliding with obj_vlad
Dying from hazards
Establishing a death state
Setting up two new events – User Defined 3 and Alarm 1
Gathering resources for hazards
Creating the sprite and object resources
Scripting scr_damage_vlad
Falling off the screen into the abyss
Fighting the player with enemies
Enemy 1 – mutant garlic
Gathering resources for garlic
Sprite resource – spr_enemy_garlic
Object resource – obj_enemy_garlic
Scripting the Create event of obj_enemy_garlic
Making garlic move with the Step event
Reacting upon collision with scr_collide_enemy
Colliding with the player – the Collision event of obj_vlad
Enemy 2 – the flying book
Gathering resources to create the flying book
The sprite resource – spr_enemy_book
The object resource – obj_enemy_book
Initializing the Create event of obj_enemy_book
Anthropomorphizing the book in the Step event
Summary
10. GOAL – Timelines and Feedback Review
GOAL!
Gathering resources for creating the goal
Sprite resource – spr_goal_door
Object resource – obj_goal_door
Introducing timelines
Using timelines
Gathering resources for integrating the timeline
Font resource – fnt_large_text
Creating and updating events for obj_goal_door
Using the Animation End event
Drawing congratulatory text with the Draw GUI event
Deactivating objects with scr_deactive_instances and the Draw

event
Freezing instances with scr_deactivate_instances
Drawing deactivated instances
Creating and applying the timeline
Step 0
Step 15
Step 90
Applying tm_finish_level
Reviewing polish, feedback, and juiciness
Gathering resources to play sounds
Playing music with scr_play_music
Reviewing the obj_particle_manager object
Giving feedback with pickups
Providing feedback when Vlad is damaged
Extending the enemy death sequence
Updating scr_collide_enemy and Step events for enemies
Creating tm_enemy_death
Step 0
Step 5
Step 25
Step 30
Summary
In closing...
Index

GameMaker Game Programming with GML

GameMaker Game Programming with GML

Copyright © 2014 Packt Publishing All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: April 2014

Production Reference: 2070514

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78355-944-2

www.packtpub.com

Cover Image by Matthew DeLucas (<mattrified@gmail.com>)

Credits

Author

Matthew DeLucas

Reviewers

Ronny Nilsson

Chris Sanyk

Chris Watts

Commissioning Editor

Edward Gordon

Acquisition Editors

Edward Gordon

Richard Harvey

Luke Presland

Content Development Editor

Dayan Hyames

Technical Editor

Veena Pagare

Copy Editors

Tanvi Gaitonde

Insiya Morbiwala

Aditya Nair

Project Coordinator

Mary Alex

Proofreaders

Stephen Copestake

Maria Gould

Ameesha Green

Linda Morris

Indexer

Mariammal Chettiyar

Production Coordinator

Melwyn D'sa

Cover Work

Melwyn D'sa

About the Author

Matthew DeLucas has been a gameplay engineer with Schell Games in Pittsburgh, Pennsylvania for over five years. He has worked on a wide range of interactive projects for PC, Web, mobiles, and consoles. Matt has also released independent projects for PC and Xbox 360, such as Convextrix, a puzzle game, and *Battle High*, which is a fighting game series. Being a programmer and designer, Matthew has also participated in almost every official, 48-hour Global Game Jam, managing to help his team achieve success while experimenting with new ideas.

Matthew began his programming career in GameMaker: Studio and has become proficient with additional game engines, such as Gamebryo and Unity3D, and scripting languages such as C#, Python, Lua, and MaxScript for 3DS Max. Often, he chronicles his experiences with game production on his blog at www.mattrifiedgames.blogspot.com.

Matthew has had a desire to work in the game industry ever since he was young, and he enjoys all of the facets of game production—programming, design, and art. His favorite genres include platformer, puzzles, racing, and fighting games, all of which influence his designs.

I wish I could list everyone I am thankful to for helping me complete this book; however, I don't think you, the reader, whom I am also thankful to, would appreciate pages of acknowledgments. Instead, I'd like to simply thank the teachers who inspired me, the friends and co-workers who give me the confidence and drive to finish this book, and most importantly, my family, for supporting me and my choice to pursue a career in the gaming industry.

About the Reviewers

Ronny Nilsson is an independent game developer with degrees in both interaction and game design. His master's degree project focused on ethical gameplay design and how it can be operationalized into an actual game. His interest in games is deep rooted, ever since his first encounter with Mario, and has always been present even when he was studying. While primarily being a designer, he also enjoys developing, especially mechanics and prototypes.

Residing in Malmö, close to Copenhagen, Ronny lives in an area where the game industry is booming; both independent and multinational game companies are expanding. Still waiting for his own first wide release, he is currently working on both commercial and experimental games in an attempt to find playful and unique gameplays.

Chris Sanyk , by day, is an IT professional with over 15 years of experience, and an indie game developer by night. Inspired by the Atari 2600 as a young child, he first started designing videogames at the age of six, and has been using GameMaker since 2010. He is an active member of the Cleveland Game Developers, Central Ohio Game Developers, and International Game Developers Association, and a regular participant in game jams such as Ludum Dare and Global Game Jam. He has been using GameMaker since 2010, and blogs and releases his projects on his website, csanyk.com.

Chris co-authored *Wireless Reconnaissance in Penetration Testing*, with *Matthew Neely* and *Alex Hammerstone*, for *Syngress*. He has been a technical reviewer for the book *HTML5 Game Development with GameMaker*, by *Jason Elliott*, *Packt Publishing*.

I'd like to thank everyone in the game development community for making the scene what it is, especially my friends and colleagues in the Ohio area: Mike Substelny, Ian Schreiber, Mike Geig, Sam Marcus, Steve Felix, Justin Demetroff, Matt Perrin, Jarryd Huntley, Brian Gessler, Eagan Rackley, Ian Faleer, Jeremy Handel, and

everyone else.

Chris Watts is a student at the University of Southampton, studying a three-year Bachelor of Science degree course in Computer Science. He first took interest in computer systems at the age of four as a result of receiving a primitive computer from his grandparents and has followed the interest ever since.

Chris has had a very inquisitive mind from a young age and was often found disassembling toys, getting his hand stuck in VCR machines, pulling every lever and pressing every button (much to his parents' dismay) in the quest to learn how everything works.

Today, Chris contributes to open source projects and invents new scripts and programs to make life easier in the digital world. He is experienced in working with over 15 programming languages and has interests in electronics, web security, design, photography, and video production.

He also offers computer/mobile repair services and web design/development services, and gives tuitions in computers to the elderly during his spare time. Chris' ambition is to make an impact in the technology world, solving problems such as interfacing with portable devices and mass adoption of cloud services.

I would like to thank, in particular, my grandmother Susan for dooming me to the domain of computers, my parents for repairing my trail of destruction around the house, and all my friends for putting up with my nerdy nature. It's all very much appreciated.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <service@packtpub.com> for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Preface

This book came about as an opportunity to create and share the knowledge of one game development enthusiast with others in the hope that they would be inspired to create their own great works. The projects in this book are not meant to be final products for readers to clone, but instead, starting points to learn basic and advanced techniques used to create games. Similarly, the code in this book shouldn't be merely copied-and-pasted but understood. Game creation is sometimes described as an exercise in problem solving. By understanding the code instead of regurgitating it, solutions will arise for dozens—if not hundreds—of possible problems, as opposed to just those introduced in this text.

GameMaker: Studio is just one of many game engines; likewise, GameMaker Language is just one of many programming languages out there. Learning these tools should not be the end of one's journey into the vast topic of game development, but instead another stone in a strong foundation, even if it is the first one.

What this book covers

Chapter 1, *Getting Started – An Introduction to GML*, introduces you to the basic formatting and syntax of **GameMaker Language** (**GML**). These topics will be expanded by creating a simple button.

Chapter 2, *Random Organization – Creating a Puzzle Game*, discusses sprite resources and randomization. A grid of puzzle pieces is created, which acts as the base for a puzzle game project.

Chapter 3, *So How Do I Play? – Adding Player Interaction*, teaches us how to add player interaction to the puzzle game using the mouse and keyboard.

Chapter 4, *Juicy Feedback – Aural and Visual Effects*, elaborates upon the implementation of sound effects and particle systems and the use of alarms, so the game can better inform the players about their progress.

Chapter 5, *Solving the Puzzle – Finishing Touches to the Puzzle Game*, helps us create a menu that allows the player to adjust various parameters of the puzzle game using the Draw events. A score and timer will also be implemented.

Chapter 6, *Finite State Machines – Starting the 2D Platformer*, starts a platformer game, focusing on the creation of the main character who is controlled by a finite state machine.

Chapter 7 , *It's in the Name – Platforms and Collisions*, expands the platformer game started in the previous chapter by adding collision through static and moving platforms.

Chapter 8, *Setting the Stage – Views, Backgrounds, and Tiles*, helps us create a camera system using views, while the platformer game's environment will be fleshed out using background resources and tiles.

Chapter 9, *Breaking Vlad – Pickups, Hazards, and Enemies*, helps the character interact with pickups to increase score and health, and also

hazards and enemies to create a challenge for the player.

Chapter 10, *GOAL – Timelines and Feedback Review*, uses timeline resources to create a way to trigger a series of events. Then, particle systems and audio will be reviewed, adding some finishing touches to the platformer.

What you need for this book

The only piece of software needed for this book is GameMaker: Studio, which can be downloaded at https://www.yoyogames.com/studio. This software will only run on Microsoft Windows systems. The older version of GameMaker can be downloaded for Mac, but some of the code may not compile in these older versions.

The projects in this book were made using version 1.2.1279 of GameMaker, with the studio license, which is free, but does have limits on the number of resources and available export options.

Who this book is for

This book is for anyone who is either learning a scripting language for the first time or for individuals who have a little experience with GameMaker and are interested in learning the scripting language instead of using the drag-and-drop icons in the hope of speeding up their game development.

Conventions

In this book, there will be a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Sprites can be drawn during the **Create** event using the built-in function, `draw_sprite`."

A block of code is set as follows:

```
var player_instance = instance_find(obj_player, 0);
player_instance.x = 100;
player_instance.y += 200;
scr_play_music(bgm_level_one, true);
```

When a particular part of a code block requires attention, the relevant lines or items are set in bold:

```
[default]
exten => s,1,Dial(Zap/1|30)
var player_instance = instance_find(obj_player, 0);
player_instance.x = 100;
player_instance.y += 200;
scr_play_music(bgm_level_one, true);
```

New terms and **important words** are shown in bold. Words seen on the screen, in menus or dialog boxes for example, appear in the text like this: "Clicking on the **Next** button, you can go to the next screen".

> ## Note
>
> Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to <feedback@packtpub.com>, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files emailed directly to you.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from https://www.packtpub.com/sites/default/files/downloads/9442OT_Images.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing

errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from http://www.packtpub.com/support.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <copyright@packtpub.com> with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at <questions@packtpub.com> if you are having a problem with any aspect of the book, and we will do our best to address it.

Chapter 1. Getting Started – An Introduction to GML

GML or **GameMaker Language** is a great tool for expanding the already vast variety of tools provided by GameMaker: Studio. GML scripts allow users to write their own code, creating an organized codebase that is easier to modify and debug than GameMaker: Studio's built-in drag-and-drop functionality.

Before exploring GML's use in creating actual games, this chapter will go over the basics of the language, such as the following components:

- Syntax and formatting
- Variables
- Functions
- Statements
- Arrays

In the second half of this chapter, many of the previously mentioned components will be used in the creation of a modular button.

Creating GML scripts

Before diving into any actual code, the various places in which scripts can appear in GameMaker as well as the reasoning behind placing scripts in one area versus another should be addressed.

Creating GML scripts within an event

Within an object, each event added can either contain a script or call one. This will be the only instance when dragging-and-dropping is required as the goal of scripting is to eliminate the need for it. To add a script to an event within an object, go to the **control** tab of the **Object Properties** menu of the object being edited. Under the **Code** label, the first two icons

deal with scripts. Displayed in the following screenshot, the leftmost icon, which looks like a piece of paper, will create a script that is unique to that object type; the middle icon, which looks like a piece of paper with a green arrow, will allow for a script resource to be selected and then called during the respective event. Creating scripts within events is most useful when the scripts within those events perform actions that are very specific to the object instance triggering the event. The following screenshot shows these object instances:



Creating scripts as resources

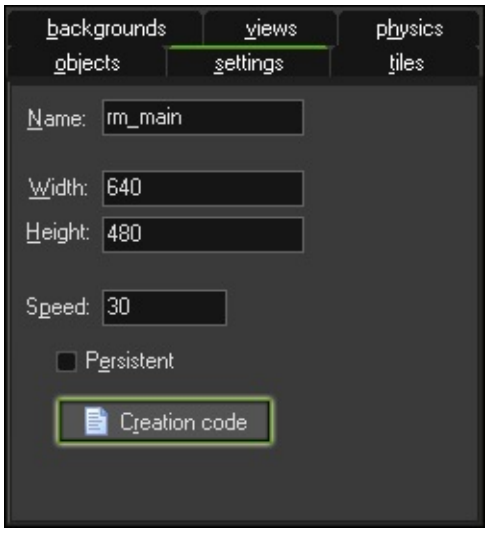
Navigating to **Resources** | **Create Script** or using the keyboard shortcut *Shift* + *Ctrl* + *C* will create a script resource. Once created, a new script should appear under the `Scripts` folder on the left side of the project where resources are located. Creating a script as a resource is most useful in the following conditions:

- When many different objects utilize this functionality
- When a function requires multiple input values or arguments
- When global actions such as saving and loading are utilized
- When implementing complex logic and algorithms

Scripting a room's creation code

Room resources are specific resources where objects are placed and gameplay occurs. Room resources can be created by navigating to **Resources** | **Create room** or using *Shift* + *Ctrl* + *R*.

Rooms can also contain scripts. When editing a room, navigate to the **settings** tab within the **Room Properties** panel and you should see a button labeled **Creation code** as seen in the following screenshot. When clicked on, this will open a blank GML script. This script will be executed as soon as the player loads the specified room, before any objects trigger their own events. Using **Creation code** is essentially the same as having a script in the **Create** event of an object.



Understanding parts of GML scripts

GML scripts are made up of many different parts. The following section will go over these different parts and their syntax, formatting, and usage.

Programs

A **program** is a set of instructions that are followed in a specific order. One way to think of it is that every script written in GML is essentially a program. Programs in GML are usually enclosed within braces, `{ }`, as shown in the following example:

```
{
    // Defines an instanced string variable.
        str_text = "Hello Word";

    // Every frame, 10 units are added to x, a built-
in variable.
    x += 10;

        // If x is greater than 200 units, the string
changes.
    if (x > 200)
    {
    str_text = "Hello Mars";
    }
}
```

The previous code example contains two assignment expressions followed by a conditional statement, followed by another program with an assignment expression.

Note

If the preceding script were an actual GML script, the initial set of braces enclosing the program would not be required.

Each instruction or line of code ends with a semicolon (`;`). This is not required as a line break or return is sufficient, but the semicolon is a common symbol used in many other programming languages to indicate the end of an instruction. Using it is a good habit to improve the overall readability of one's code.

Tip

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

snake_case

Before continuing with this overview of GML, it's very important to observe that the formatting used in GML programs is **snake case**. Though it is not necessary to use this formatting, the built-in methods and constants of GML use it; so, for the sake of readability and consistency, it is recommended that you use snake casing, which has the following requirements:

- No capital letters are used
- All words are separated by underscores

Variables

Variables are the main working units within GML scripts, which are used to represent values. Variables are unique in GML in that, unlike some programming languages, they are not strictly typed, which means that the variable does not have to represent a specific data structure. Instead,

variables can represent either of the following types:

- A number also known as `real`, such as 100 or 2.0312. Integers can also correspond to the particular instance of an object, room, script, or another type of resource.
- A `string` which represents a collection of alphanumeric characters commonly used to display text, encased in either single or double quotation marks, for example, `"Hello World"`.

Variable prefixes

As previously mentioned, the same variable can be assigned to any of the mentioned variable types, which can cause a variety of problems. To combat this, the prefixes of variable names usually identify the type of data stored within the variable, such as `str_player_name` (which represents a string). The following are the common prefixes that will be used throughout this book:

- **str**: String
- **spr**: Sprites
- **snd**: Sounds
- **bg**: Backgrounds
- **pth**: Paths
- **scr**: Scripts
- **fnt**: Fonts
- **tml**: Timeline
- **obj**: Object
- **rm**: Room
- **ps**: Particle System
- **pe**: Particle Emitter
- **pt**: Particle Type
- **ev**: Event

Note

Variable names cannot be started with numbers and most other non-alphanumeric characters, so it is best to stick with using basic letters.

Variable scope

Within GML scripts, variables have different scopes. This means that the way in which the values of variables are accessed and set varies. The following are the different scopes:

- **Instance**: These variables are unique to the instances or copies of each object. They can be accessed and set by themselves or by other game objects and are the most common variables in GML.
- **Local**: Local variables are those that exist only within a function or script. They are declared using the `var` keyword and can be accessed only within the scripts in which they've been created.
- **Global**: A variable that is global can be accessed by any object through scripting. It belongs to the game and not an individual object instance. There cannot be multiple global variables of the same name.
- **Constants**: Constants are variables whose values can only be read and not altered. They can be instanced or global variables. Instanced constants are, for example, `object_index` or `sprite_width`. The `true` and `false` variables are examples of global constants. Additionally, any created resource can be thought of as a global constant representing its ID and unable to be assigned a new value.

The following example demonstrates the assignment of different variable types:

```
// Local variable assignment.
var a = 1;

// Global variable declaration and assignment.
globalvar b;
b = 2;

// Alternate global variable declaration and
assignment.
global.c = 10;

// Instanced variable assignment through the use of
```

```
"self".
self.x = 10;

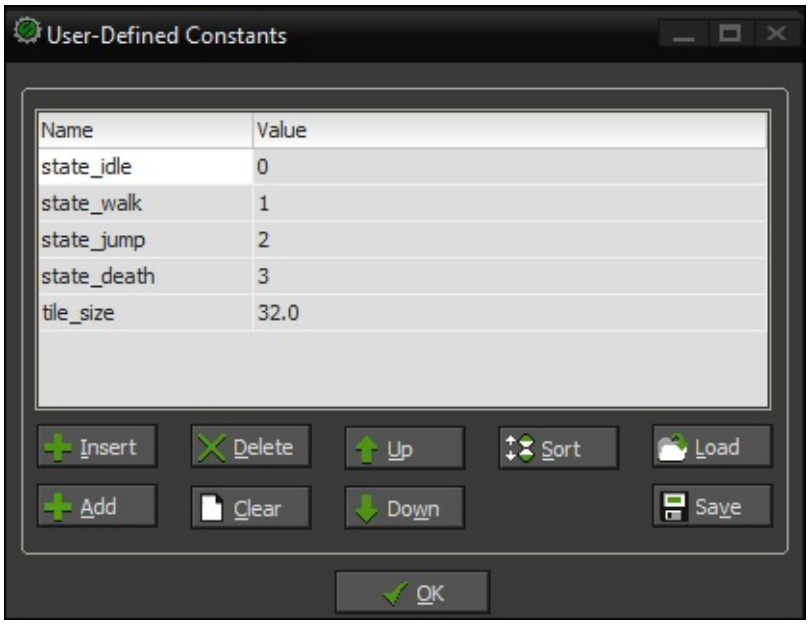
/* Instanced variable assignment without the use of
"self".  Works identically to using "self". */
y = 10;
```

Built-in variables

Some global variables and instanced variables are already provided by GameMaker: Studio for each game and object. Variables such as `x`, `sprite_index,` and `image_speed` are examples of built-in instanced variables. Meanwhile, some built-in variables are also global, such as `health`, `score`, and `lives`. The use of these in a game is really up to personal preference, but their appropriate names do make them easier to remember. When any type of built-in variable is used in scripting, it will appear in a different color, the default being a light, pinkish red. All built-in variables are documented in GameMaker: Studio's help contents, which can be accessed by navigating to **Help** | **Contents...** | **Reference** or by pressing *F1*.

Creating custom constants

Custom constants can be defined by going to **Resources** | **Define Constants...** or by pressing *Shift* + *Ctrl* + *N*. In this dialog, first a variable name and then a correlating value are set. By default, constants will appear in the same color as built-in variables when written in the GML code. The following screenshot shows this interface with some custom constants created:



Functions and accessing script resources

A function is a statement that executes a program; it is either built into GML or created as a script resource. Functions can either execute an action, such as changing the alignment of a font during a **Draw** event, return a value, or do both. Functions have to be followed by a set of parentheses—`( )`—to execute properly. Another important aspect of functions is **arguments**. These comma-separated sets of data—string, integers, objects, and so on—are accessible to functions when executed. If there are no arguments, the parentheses are left empty; however, when needed, arguments are placed in between them. The following are some examples of functions with and without arguments:

```
// Executes an action, in this case, drawing the
instance.
draw_self();

/* Executes an action which requires arguments, in
this case, drawing an arrow. */
draw_arrow(0,0,100,100,2);

// Obtains a random value between two provided values.
random_value = random(10, 23);
```

GameMaker: Studio provides a wide variety of functions. Many of these functions will be used throughout this book; however, to find information about all of GML's available functions, go to **Help** | **Contents...** | **Reference** or press *F1*.

Scripts created as resources can be accessed using two methods. Either the script's name can be referenced and used like a built-in GML function or the function `script_execute` can be used, as shown in the following code snippet:

```
// Executes the script directly like a built-in
function.
scr_script_resource("argument", obj_button, 0.12,
false);

/* Executes the same script as the previous line but
through the use of "script_execute". */
script_execute(scr_script_resource, "argument",
obj_button, 0.12, false);
```

The advantage of using `script_execute` is that it allows a script-assigned variable to be used as shown in the following code:

```
// Assigns an instanced variable with the script
resource's index.
self.script = scr_script_resource;

// Executes the assigned script.
script_execute(self.script, "argument", obj_button,
0.12, false);
```

The `script_execute` function can only be used on scripts created as resources; additionally, variables cannot be assigned built-in functions. The following code demonstrates this problematic situation:

```
// Assigns an instanced variable with a script
resource ID.
self.script = scr_script_resource;

// Calling the instanced variable will cause a compile
```

```
error.
self.script("argument", obj_button, 0.12, false);
```

Arguments

As mentioned previously, some functions require arguments. When creating a script, these arguments can be accessed within the script using the keywords `argument0` through `argument15`, allowing for up to 16 different values if necessary.

Expressions

Expressions represent values usually stored within variables or evaluated by a conditional statement, which will be explained later. They can be real numbers, such as 3.4; hexadecimal numbers starting with a `$` sign, such as `$00FFCC` (usually used to represent a color); and strings, which are created by encasing them in single or double quotation marks, for example, `'hello'` or `"hello"`.

Expression symbols

Expressions can be manipulated and evaluated using different symbols. The equals sign or assignment operator `=` sets the value of a variable as shown in the following code:

```
// Assigning a variable with a value.
a = 10;

/* Assigning a different variable with an expression,
in this case, the sum of the previously declared
variable and 7.5. */
b = (a + 7.5);
```

Expressions can also be combined with basic mathematical operations, as shown in the following code:

```
// Addition and subtraction, + and -
val = a + 20 - b;
```

```
// Multiplication and division, * and /
val = a  20  b;

 Expressions encased in parenthesis, ( and ), will be
evaluated first. /
val = (a + 20)  (b - 40);

// + can also be used to concatenate, or link, strings
together.
str = "hello " + "world";
```

Mathematical operations can be combined with `=` to create a compound assignment operator and perform relative addition, subtraction, multiplication, or division, as shown in the following code:

```
// Relative addition, +=
x += y; // equivalent to x = x + y;

// Relative subtraction, -=
x -= y; // equivalent to x = x – y;

// Relative multiplication, =
x = y; // equivalent to x = x * y;

// Relative division, =
x = y; // equivalent to x = x / y;
```

The main advantage of this code is its extreme simplicity. In the previous examples, `x` is a simple variable to type out, but if a variable name is longer, the preceding code cuts down on having to unnecessarily retype that name on both sides of the assignment operator.

Variables can also be incremented by one value as shown in the following code:

```
var a, b, c, d, str_a, str_b, str_c, str_d;
a = 1;
b = 1;
c = 1;
d = 1;

// The return string will be "1" but a's value is now
```

```
2.
str_a = string(a++);

// The return string will be "2" and b's value is 2.
str_b = string(++b);

// The return string will be "1" but c's value is 0;
str_c = string(c--);

// The return string will be "0" and d's value is 0;
str_d = string(--d);
```

In summary, if `++` or `--` is included after the variable, the variable's current value is returned and then is incremented. If `++` or `--` is set before the variable, its current value is incremented and that new value is returned.

Boolean comparisons, as shown in the following code, compare expressions and return the values `true` or `false`, which are GML constants equal to `1` and `0` respectively:

```
// Less than, <
if (a < 20) { instance_create(a, 20, obj_ball); }

// Less than or equals to, <=
if (a <= 20) { instance_create(a, 20, obj_ball); }

// Equals to, ==
if (a == 20) { instance_create(a, 20, obj_ball); }

// Not equals to, !=
if (a != 20) { instance_create(a, 20, obj_ball); }

// Greater than, >
if (a > 20) { instance_create(a, 20, obj_ball); }

// Greater than or equals to, >=
if (a >= 20) { instance_create(a, 20, obj_ball); }
```

Booleans can also be combined for evaluation, as shown in the following code:

```
// And, &&, will return true if both booleans are also
true.
```

```
if (a == 20 && b == 40) { val = "and"; }

/* Or, ||, will return true if at least one of the
booleans is true.
if (a == 20 || b == 40) { val = "or"; }

 xor, ^^, will return true if one booleans is true and
the other false. */
if (a == 20 ^^ b == 40) { val = "xor"; }
```

Conditional statements

Conditional statements utilize a Boolean expression with a corresponding program. If the value returned is `true`, the program following the conditional statement will be executed; otherwise, it will be skipped. There are several types of conditional statements in GML, each of which has its own uses.

if, else, and switch

The `if` statement is probably the most common conditional statement that will be used while making these games. The `if` statements were introduced when discussing Booleans previously; the following is another example illustrating a simple `if` statement:

```
/* If the value of a is greater than the value of b, a
is returned; otherwise, b is returned. */
if (a > b)
{
  return a;
}
else
{
  return b;
}
```

In the previous example, assuming both `a` and `b` are real numbers, if the value of `a` is greater than that of `b`, `a` is returned; otherwise, `b` is returned. One thing that should be noted is that `b` will be returned if it is *less than or equal to* `a`, since this is the opposite of *greater than*.

Now what if a variable needs to be compared against many different conditions? The following *could* be done:

```
/* Assigns the temperature of an object based on a
color with multiple if-else-statements. */
if (color == c_green || color == c_purple || color ==
c_blue)
{
    temperature = "cool";
}
else if (color == c_red || color == c_orange || c ==
c_yellow)
{
    temperature ="warm";
}
else if (color == c_black || color == c_gray || color
== c_white)
{
    temperature ="neutral";
}
else
{
    temperature ="other";
}
```

When there are a lot of objects, each with its own complex program that needs to be executed, the execution of these objects could become rather tedious and confusing. Instead of using long chains of `if-else` statements, a `switch` statement can be used. A `switch` statement is created with the keywords `switch`, `case`, `break`, and `default`, as shown in the following code:

```
/* Assigns the temperature of an object based on a
color with a switch statement. */
switch (color)
{
  case c_red:
  case c_orange:
  case c_yellow:
    temperature = "warm";
    break;
  case c_green:
  case c_blue:
```

```
    case c_purple:
      temperature ="cool";
      break;
    case c_black:
    case c_white:
    case c_gray:
      temperature = "neutral";
      break;
    default:
      temperature = "other";
      break;
}
```

The previous `switch` statement is identical to the layered `if-else` statements created earlier. Each `case` statement is essentially testing whether or not the value of the variable supplied after `switch` is equal to its own. If the values are equal, the program is executed. If a `default` case is supplied, this program will be executed if none of the other cases satisfy the conditional statement. The keyword `break`, which will be discussed again in a later section, is used to end the program running within a `case` statement.

repeat, while, do, and for

The `repeat`, `while`, `do`, and `for` statements are all examples of statements that execute a program multiple times and this is often referred to as a **loop**. The `Repeat` statement is used to execute the same program for a specified number of times:

```
// Creates 10 buttons at random positions between 0
and 100.
repeat (10)
{
  instance_create(random(100), random(100),
obj_button)
}
```

The previous code will create 10 instances of `obj_button` at random positions. The `while` statement will execute a program until a condition is no longer met:

```
// Reduces x by 10 until it is no longer greater than
100.
while (x > 100)
{
  x -= 10;
}
```

In the previous code, the value of `x` will be reduced until it is less than 100. The `do` statements are very similar to the `while` statements, except they require an additional keyword—`until`:

```
/* Initially reduces x by 10 but continues until x is
less than or equal to 100. */
do
{
  x -= 10;
}
until (x <= 100);
```

The difference between `do` and `while` is subtle. In a `do` statement, the program is executed first and then the condition is checked; in a `while` statement, the condition is checked first before executing the program. The `for` statements or `for` loops have built-in conditions as shown in the following code:

```
// Declares a local variable.
var i;

/* 10 buttons are created, evenly spaced apart
horizontally by 100 units with an initial offset of
25. /
for (i = 0; i < 10; i++)
{
    instance_create(i  100 + 25, 25, obj_button)
}
```

The first statement after the keyword `for` sets the local variable `i` to `0`. Then, there is the condition that if the value of `i` is less than `10`, the block of code within the loop is executed. The value of the variable `i` is then incremented and checked again within the conditional statement, running the program if the statement is still valid. Now the previous `for` statement

is very similar to the `repeat` statement. The important difference is that `i` is incremented by the statement, making this a good way to loop through values in a one-or two-dimensional array, which will be covered shortly. The following sample code demonstrates this functionality with a two-dimensional array:

```
// Creates two local variables.
var i,j;

/* This is an example of a nested for loop in which a
10 by 10 grid with values ranging from 0 to 99 will be
created. /
for (i = 0; i < 10; i++)
{
  for (j = 0; j < 10; j++)
  {
    grid[i,j] = i + j  10;
  }
}
```

The following code shows the same functionality as the one performed using the `repeat` statements:

```
// Creates local variable and then assigns their
values to 0.
var i, j;
i = 0;
j = 0;

/* Repeat functionality is performed 10 times and 10
more times within each repeat-statement. /
repeat (10)
{
  repeat(10)
  {
    // assign the grid value
    grid[i,j] = i + j  10;

    // increment after each inner repeat.
    j++;
  }
  // increment after each outer repeat.
  i++;
}
```

As demonstrated, a `for` statement is much clearer than a `repeat` statement in situations like these.

Before moving on, it must be noted that the `repeat`, `while`, `do`, and `for` statements can create a situation known as the **infinite loop**. Infinite loops are very dangerous and cause a game to freeze up, ignoring input from the user and forcing them to shut down the game. The following are some examples of statements that can cause this; *do not* use these in actual code as they are just examples of what not to do:

```
/* Since the condition in the while-statement is
always true, it will never terminate. /
while (true)
{
  x += 1;
}

// Declares a local variable.
var i;

/ This for-statement will never end since i will never
be less than 0. */
for (i = 0; i >= 0; i++)
{
  x += i;
}
```

break, continue, and return

As previously mentioned, looping statements can sometimes cause a situation in which they do not terminate. There are two ways to resolve this issue. One is known as `break`, which was already demonstrated in the `switch` statements to break out of a `case` statement, but it can also be used to break out of a loop. The following is an example of a `break` statement:

```
// local variable used in the for loop
var i;

/* Iterates through 1000 enemies.  At the first
```

```
undefined enemy, the loop is exited, otherwise, the
enemy moves by a random value between 1 and 10
horizontally. */
for (i = 0; i < 1000; i++)
{
  if (enemies[i] == noone)
  {
    break;
  }
  enemies[i].x += random_range(1,10);
}
```

In the previous code sample, the `for` statement is broken out of and no longer continues running once the internal condition is met. Suppose `break` were to be replaced with `continue`, as shown in the following example:

```
// local variable for for loop
var i;

/* Iterates through 1000 enemies.  If an undefined
enemy is encountered, the remaining enemies are still
checked; otherwise, the enemy is moved by a random
value between 1 and 10 horizontally. */
for (i = 0; i < 1000; i++)
{
  if (enemies[i] == noone)
  {
    continue;
  }
  enemies[i].x += random_range(1,10);
}
```

The `for` statement would proceed to checking the additional enemies, even if they are all undefined. The `break` and `continue` statements, however, are very useful because in both examples, an undefined enemy is to be encountered and the game will crash upon trying to add the value generated by `random_range` to its `x` value. The `return` statements are used most often in GML scripts that are created as resources. The `return` statement will not only end any script in its entirety, but also return whatever value follows it as demonstrated in the following code:

```
// This script will simply add and multiply two
arguments
var result;
result = (argument0 + argument1) * (argument0 *
argument1);
return result;
```

If you want to obtain a value using the previously shown function, it can be accessed through the following code, assuming that the function `scr_add_multipy` is a script resource:

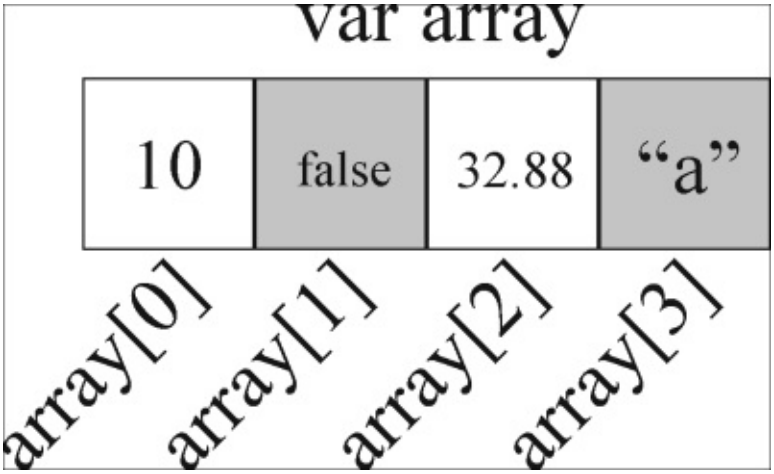
```
// Execute the method and assign it to x.
x = scr_add_multiply(1, 2);
```

Arrays

Now the one additional feature that variables have is that they can contain multiple values through the use of **arrays**. An array is a fundamental and useful programming tool. Instead of having to create a unique variable name to store dozens of values, such as those for a high-score list or a role-playing game's inventory system, all of these values can simply be stored in an array. The following code demonstrates the creation and assignment of an array in code:

```
// A local array is declared and assigned four values.
var array;
array[0] = 10;
array[1] = false;
array[2] = 32.88;
array[3] = "a";
```

The following figure illustrates the data contained within each portion of the newly created variable:



By enclosing an integer within brackets—`[ ]`—after a variable name, a one-dimensional array is created. This integer is then called an **index**.

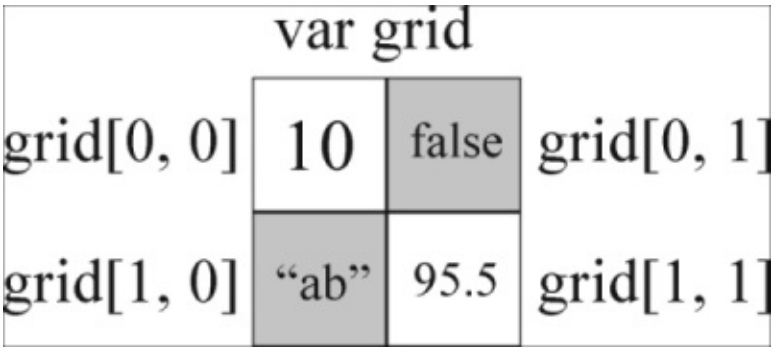
Arrays are useful for storing data that is iterated, but they do have their limits. An array cannot contain more than 32,000 members as this will cause an error, and arrays with several thousand members can cause memory issues and their initialization can be rather slow, so it is best to avoid using them when possible.

Two-dimensional arrays

A two-dimensional array is created by adding two comma-separated integers within the brackets after a variable name. They are great to use when creating a grid system: the first index represents a column and the second represents a row, as shown in the following code:

```
/* A 2-dimensional array is declared and assigned four
different values. */
var grid;
grid[0,0] = 10;
grid[0,1] = false;
grid[1,0] = "ab";
grid[1,1] = 95.5;
```

The following figure illustrates the data within the two-dimensional array created in the previous code block:



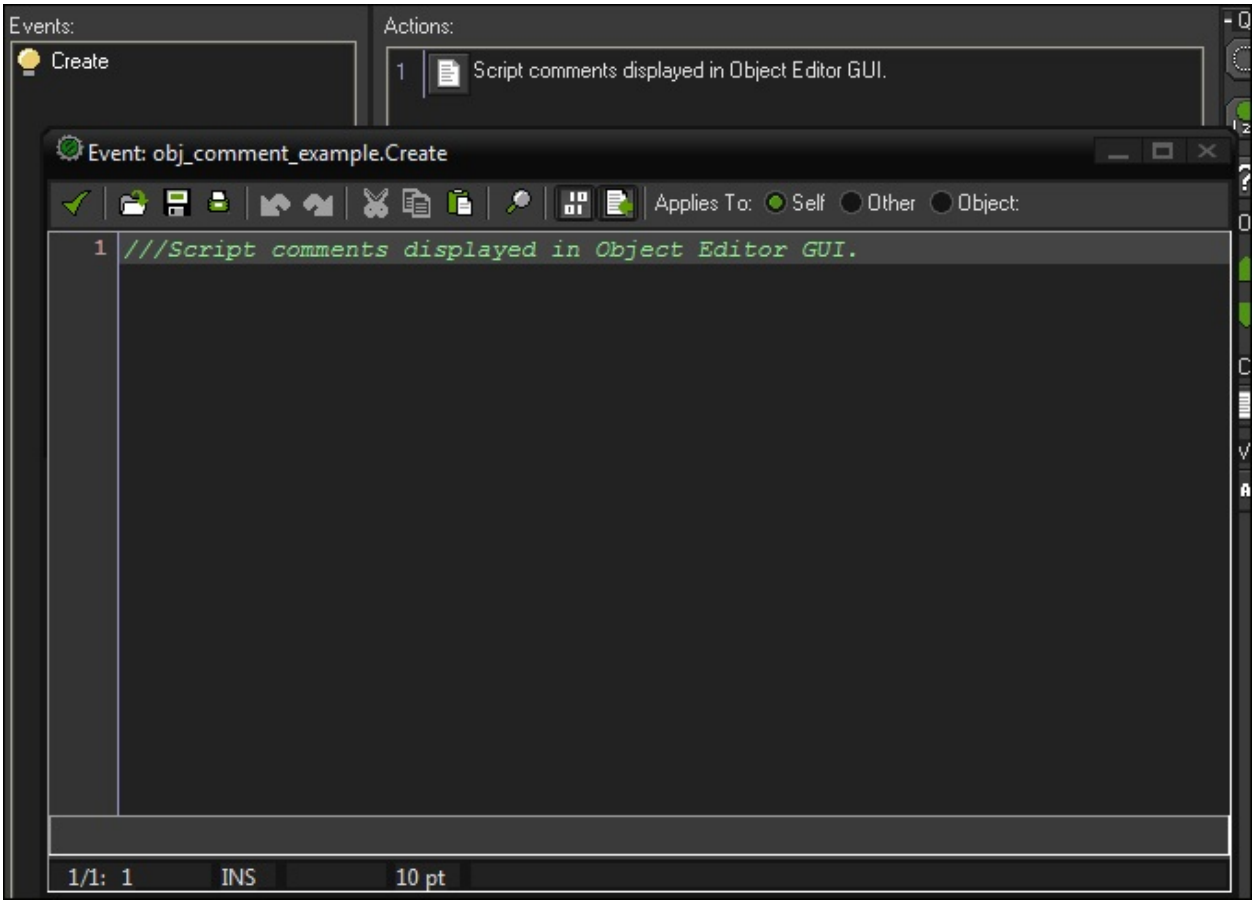
Commenting

Throughout the previous code samples, characters such as `//`, `/*`, or `*/` have been used. These create **comments**, which are lines and blocks of code that are not compiled or that will not show up in the results of the game. Comments are used mostly to document code so that when others are looking at that code, it is easier to understand it and appears much more professional. They can also help the original programmer by citing information that may not be obvious or when the code hasn't been looked at in quite some time. Comments can also be used to temporarily remove a section of code without having to delete it while debugging or testing new features.

Note

Up to this point, the comments in the provided code samples have been very verbose for demonstration purposes. In future sections, the comments will not be as verbose, but will still be provided so that the examples are clear and understandable.

The `//` characters will comment just one line of code. Any number of lines encased within `/*` and then `*/` will be part of a **comment block**. Finally, using `///` in the first line of a script used in an event will make the comment show up in the editor as shown in the following screenshot:



Errors

No one is perfect and mistakes are often made when writing scripts of any kind. There are two types of errors that are encountered when working in GameMaker: Studio—**compile errors** and **runtime errors**.

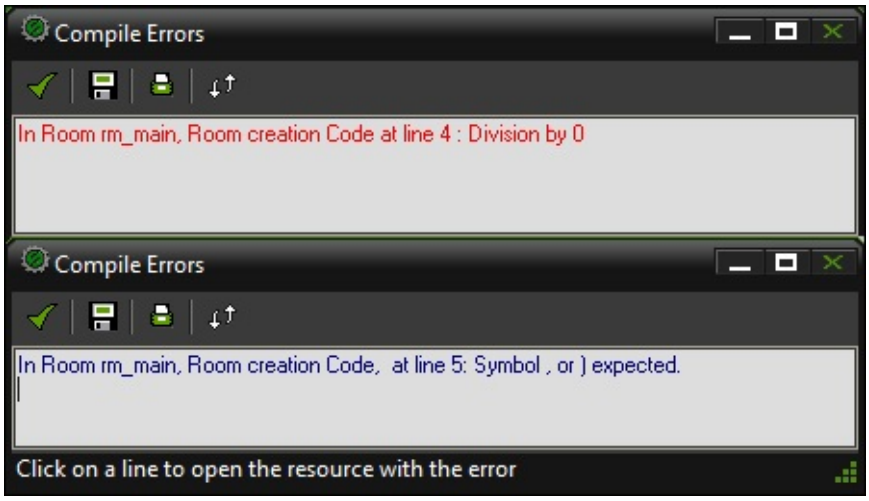
A compile error is one that prevents GameMaker: Studio from building the project. If a compile error occurs, the game cannot even start. The common reasons for compile errors are as follows:

- Omitting a closing parenthesis, single quotation mark, double quotation mark, or bracket for every opened one
- Misspelled variables, functions, or script names
- Accessing an argument variable that has not been provided in a function call
- Omitting commas between arguments in a function

A runtime error is one that causes a game to crash while it is active. Such errors can happen for any number of reasons. The following are a few examples:

- Incorrect data provided as an argument in a function
- Attempting to access instanced variables of an undefined object
- Accessing an undefined member of an array
- Division by 0

If either of these types of errors occur while working on a game in GameMaker: Studio, a window will pop up describing the error and where it is occurring. Though the name of the window in GameMaker: Studio will be titled **Compile Errors**, runtime errors will be displayed in red text and compile errors will be navy blue by default, as shown in the following screenshot:

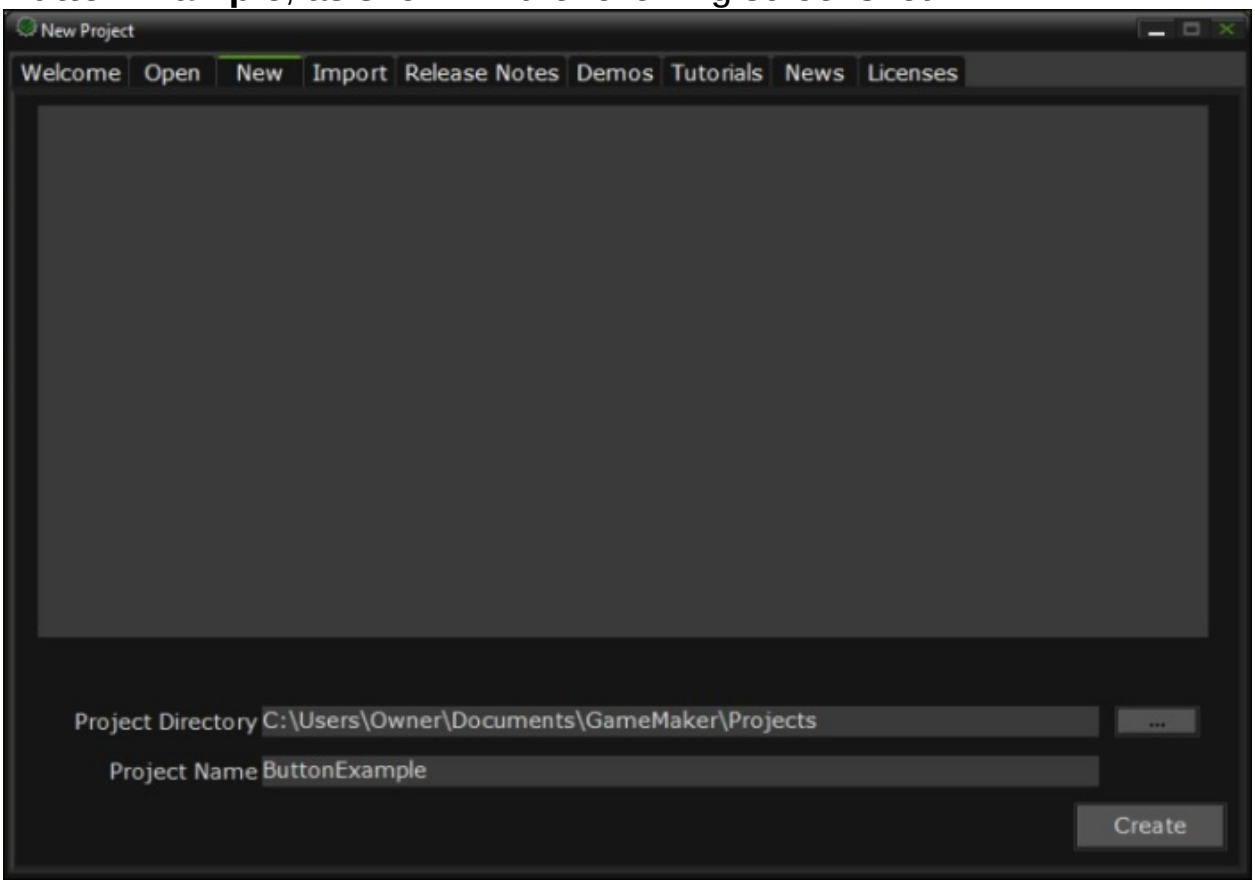

Compile Errors
In Room rm_main, Room creation Code at line 4 : Division by 0
Compile Errors
In Room rm_main, Room creation Code, at line 5: Symbol , or) expected.
Click on a line to open the resource with the error

Pushing your buttons

Now that the basic definitions and the syntax of GML have been covered, a simple button will be created to demonstrate some of these concepts. Buttons are a crucial part of most games' user interfaces and have a lot of uses. This button can be re-used in future projects, which allows the creation of custom buttons with just a few lines of code.

Creating the project

Before starting, a new project must be created. Upon opening GameMaker: Studio, a dialog window will open with several tabs. Click on the **New** tab to create a new project. This project will be named **ButtonExample**, as shown in the following screenshot:



Gathering resources

In this section, many of the different assets—objects, sprites, rooms, and so on—that are needed and referenced in the scripts as well as their attributes will be created. These attributes—size, scale, and so on—do not have to be identical to the ones created here, but will help with maintaining consistency. It is suggested that before writing any of the scripts in this section, all of the required resources be created. After right-clicking on the folder of each resource type, the option to create a new resource type is available. The different types of resources can also be created in the following ways:

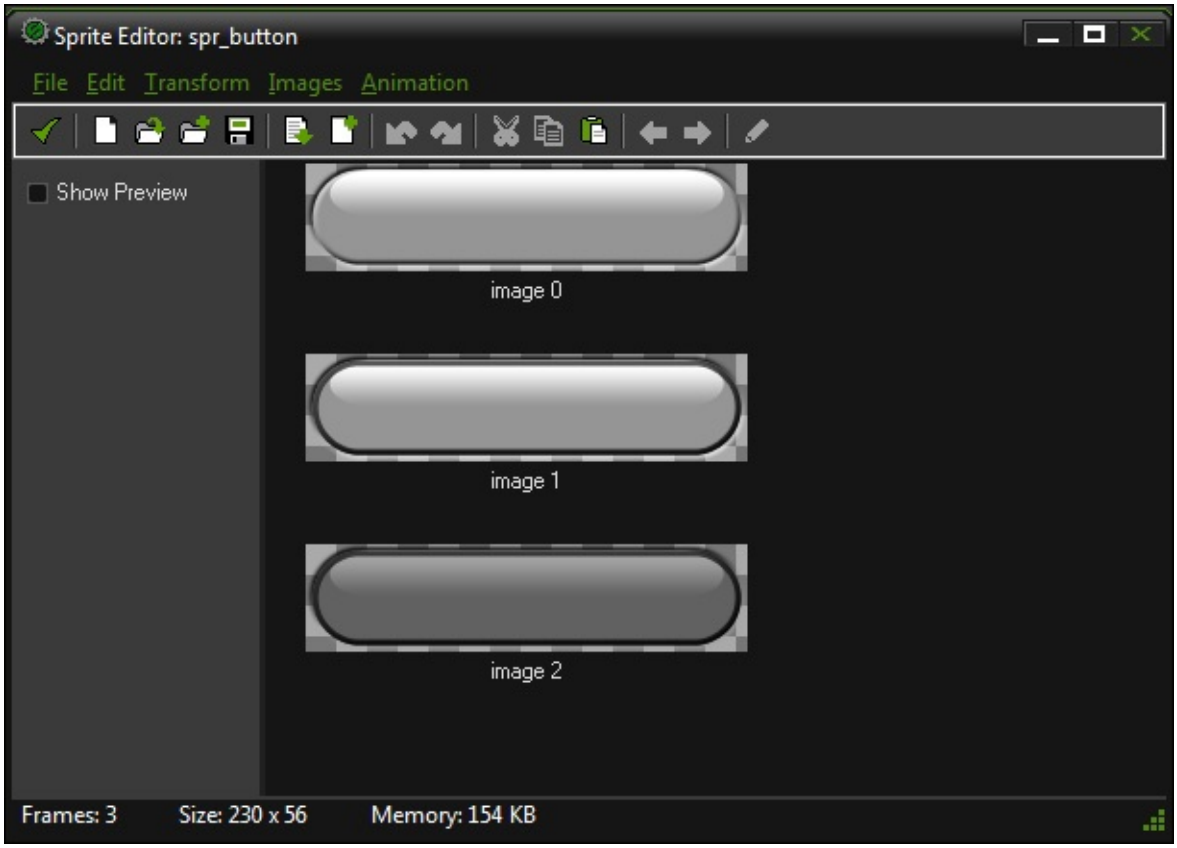
- A sprite can be created by navigating to **Resource** | **Create Sprite** or by pressing *Ctrl* + *Shift* + *S*
- The font can be created by navigating to **Resource** | **Create Font** or by pressing *Ctrl* + *Shift* + *F*
- The timeline can be created by navigating to **Resources** | **Create TimeLine** or by pressing *Ctrl* + *Shift* + *T*
- The background can be created by navigating to **Resources** | **Create Background** or by pressing *Ctrl* + *Shift* + *B*
- The path can be created by navigating to **Resources** | **Create Path** or by pressing *Ctrl* + *Shift* + *P*
- The script can be created by navigating to **Resources** | **Create Scrip**t or by pressing *Ctrl* + *Shift* + *C*
- The sound can be produced by navigating to **Resources** | **Create Sound** or by pressing *Ctrl* + *Shift* + *U*

Additional tips on creating and editing assets can be found in GameMaker: Studio's help menu, which can be accessed through **Help** | **Contents...** | **Reference** or by pressing *F1*.

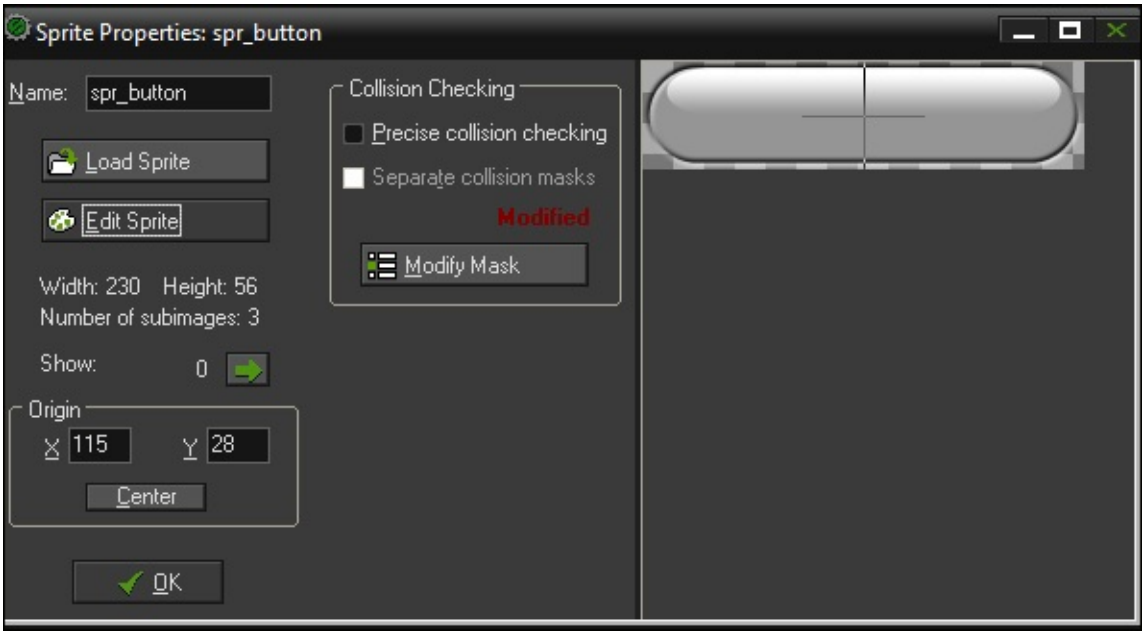
Sprites – spr_button

Only one sprite is needed to create this button. There are three important aspects of sprites: **subimages**, **origin**, and **bounds**. First, the subimages or frames of what will actually be displayed should be defined. By clicking on the **Edit Sprite** button, the **Sprite Editor** window will open. The button will be made from three frames representing three different states: a default state, a rollover state, and a down state, all of which are

shown in the following screenshot. These sprites can be added using the images inside the `sprites` folder for this chapter's code files.



Once the sprites are loaded, the origin can be defined. The origin of the `spr_button` sprite is set to its center. This origin determines how the button is offset from its position. An origin of (0, 0) would align the button to the upper-left of its current position. This setting is displayed in the following screenshot with a crosshair on the sprite:



Finally, the bounding box of `spr_button` can be defined. The bounding box defines an area that the player will be able to interact with. This sprite will use a rectangular bounding box with an **alpha tolerance** of `128`. The alpha tolerance determines how much of the sprite's transparency will be used to determine the definition of the automatic bounding box. The lower the value, the more of the sprites' alpha will be used—`0` means that the entire image will be used and `255` means that the entire bounding box invalid. These settings are displayed in the following screenshot:

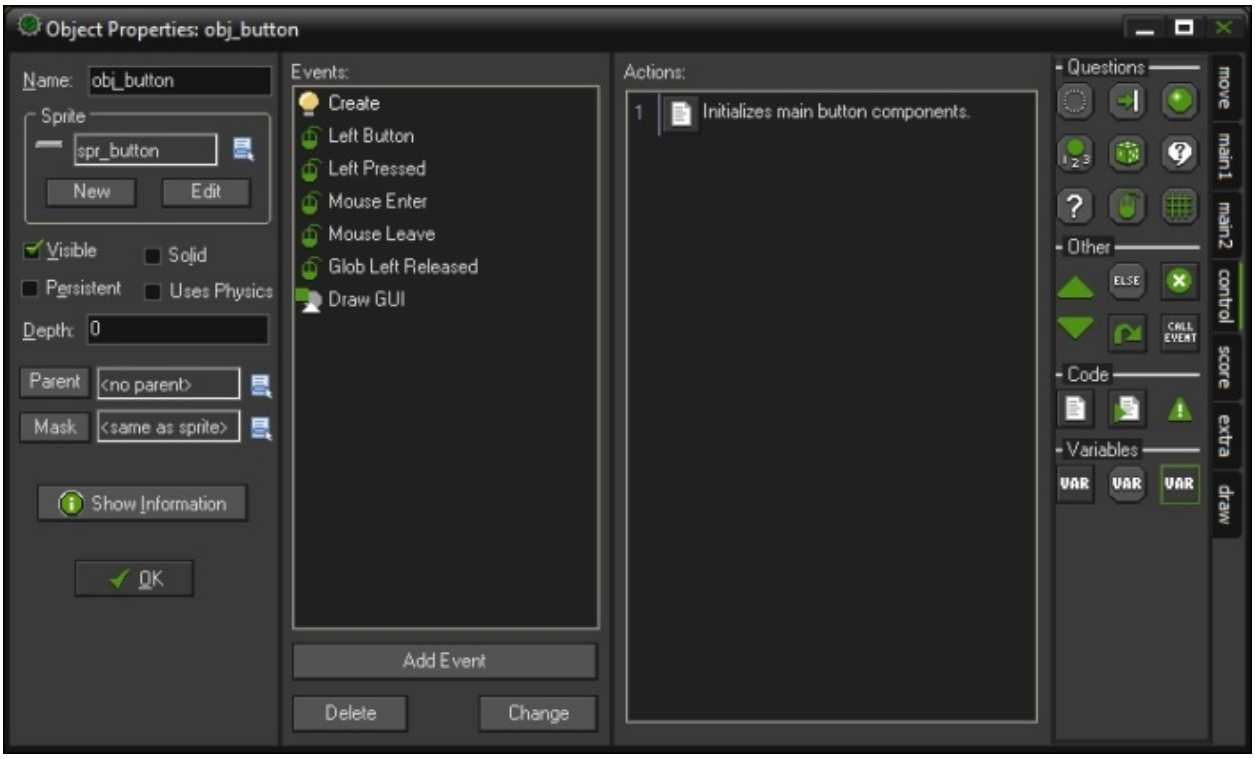


Objects – obj_button

The `obj_button` object will represent the buttons that are going to be created. This object needs certain events that can be added by navigating to the following paths:

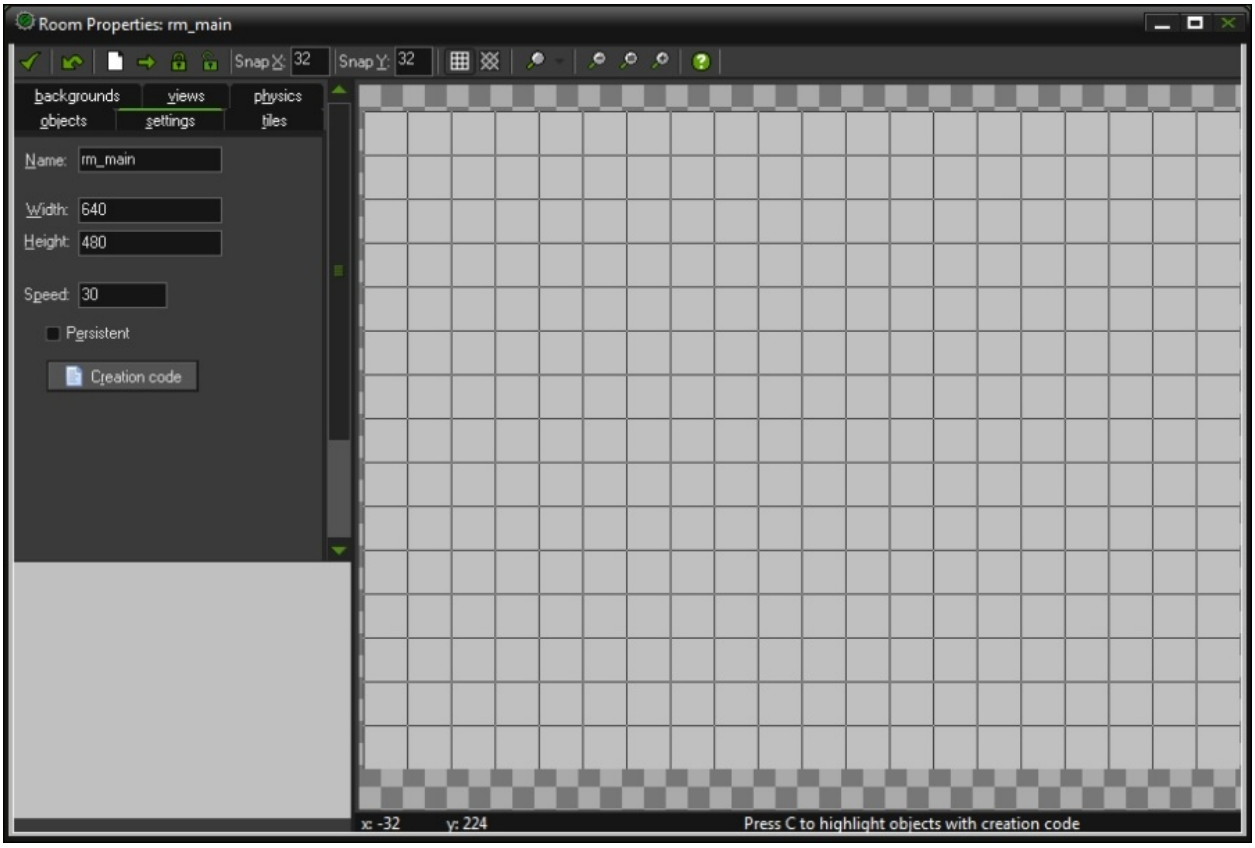
- **Add Event | Create**
- **Add Event | Mouse | Left button**
- **Add Event | Mouse | Left pressed**
- **Add Event | Mouse | Mouse enter**
- **Add Event | Mouse | Mouse leave**
- **Add Event | Mouse | Global mouse | Global left released**
- **Add Event | Draw | Draw GUI**

These different events will be explained and scripted later when creating the actual button. The sprite of `obj_button` should be set to `spr_button`, as illustrated in the following screenshot:



Room – rm_main

All games made within GameMaker need at least one room to run. This room is set to be 640 pixels wide and 480 pixels tall, but its size is irrelevant for this example. Additionally, the creation code will be used in this room, so do not add any instances of `obj_button` to it as one will be created with the code. The following screenshot shows the **Room Properties** page:



The events of obj_button

As discussed earlier, `obj_button` contains many different events, but those events are all relatively simple. To add an event to an object, click on the **Add Event** button at the bottom of the **Events** panel in the object editor window. In this section, the code for each event and the reasoning behind it will be explained.

The Create event

The **Create** event (**Add Event** | **Create**) is the first program executed when a new instance of an object is created. To create the event of `obj_button`, an **Execute Code** icon should be dragged-and-dropped in from the **control** tab of the **Object Properties: obj_button** panel. In the following script, the instantiation of the button's main variables will be handled:

```
/// Initializes main button components.

// Sets the image speed of the button to 0 so it will
not animate.
image_speed = 0;

// Sets the image index to the first frame of the
sprite set.
image_index = 0;

// Is this button currently down?
is_down = false;

// Is the mouse currently over the button?
is_over = false;

// Assigns the displayed text.
str_text = "Hello World";

// Scripts called during the different events.
// The indices in this array are built-in constants
that represent // different values and used for
clarity. */

scripts[ev_left_button] = noone;
scripts[ev_left_press] = noone;
scripts[ev_mouse_enter]  = noone;
scripts[ev_left_release] = noone;
```

Note

The constant `noone` is used to indicate that a variable is undefined, but `noone` is actually equal to `-4` since the concept of null or undefined objects does not truly exist in GML.

The Left Button event

The **Left Button** event (**Add Event** | **Mouse** | **Left button**) will be executed for every frame for which the left button of the user's mouse is down and the position of the mouse is within the position of the object's sprite's bounds. The following code will be executed when the button

triggers this event:

```
/* Executes the script when the button is being held
down with the left mouse. */
if (scripts[ev_left_button] != noone)
{
    script_execute(scripts[ev_left_button]);
}
```

The previous code is an example of a conditional statement. The script at the index `ev_left_button` is then checked. If the index is assigned, the script will be executed.

The Left Pressed event

The code for the **Left Pressed** event (**Add Event** | **Mouse** | **Left pressed**) will have a bit more functionality than the **Left Button** event. Unlike the **Left Button** event, the **Left Pressed** event will be executed only once while the user's mouse is clicked on when the cursor is within the button's bounds. The following code is executed when the specified script is assigned:

```
// If the specified script is assigned, it is
executed.
if (scripts[ev_left_press] != noone)
{
    script_execute(scripts[ev_left_press]);
}

// Indicates that the button is down.
is_down = true;

// Sets the image index to the down state image index.
image_index = 2;
```

The previous code is similar to that of the **Left Button** event except for two minor differences. Firstly, a different index is used when testing for the presence of a script in the initial condition. Secondly, the `is_down` Boolean is set to `true` indicating that the button has been pressed, and `image_index` is set to `2`, using the third frame of the sprite's animation.

The Mouse Enter event

Most buttons have a rollover state, indicating that the button is in the range of being clicked. The populating of the **Mouse Enter** event (**Add Event** | **Mouse** | **Mouse enter**) will be scripted similarly to the other events used thus far, as shown in the following code:

```
// Indicates that the mouse is currently over the
button.
is_over = true;

// If the specified script is assigned, it is
executed.
if (scripts[ev_mouse_enter] != noone)
{
    script_execute(scripts[ev_mouse_enter]);
}

/* If the left-mouse button is not down, the image
index is assigned to the rollover frame. */
if (!is_down)
{
    image_index = 1;
}
```

The `is_over` object is set to `true` since the user's cursor is now hovering over the button. Then, the script indexed at `ev_mouse_enter` is tested and performed if assigned. Finally, the value of `is_down` is also tested. If the value is `true`, the button should stay in the down frame of its animation and not go to the rollover stage.

The Mouse Leave event

When the user's mouse leaves the bounds of the button, the **Mouse Leave** event (**Add Event** | **Mouse** | **Mouse leave**) will be triggered. This needs to be done to indicate that the cursor is no longer hovering over this button. The button should go to the first frame of its animation; however, this is only if the user no longer has the left mouse button held down. The following code explains the syntax for this event:

```
// Indicates that the mouse is no longer over the
button.
is_over = false;

/* If not being held down, the image index is assigned
to the default state frame. */
if (!is_down)
{
    image_index = 0;
}
```

The Global left release event

The **Global left release** event (**Add Event** | **Mouse** | **Global mouse** | **Global left released**) will be triggered regardless of the user's position when the left mouse button is released. The releasing function, however, will only be executed if the cursor is currently hovering over the button. The button should always return to the first frame or the default state frame, which is why a **Global left release** event is used as opposed to a **Left Release** event. The following code gives the syntax for this event:

```
// Indicates that the button is no longer being
pressed.
is_down = false;

/* If the mouse is over the button, the release script
is executed and the image index is set to the over
frame; otherwise, the image index is set to the
default frame. */
if (is_over)
{
    image_index = 1;
    if (scripts[ev_left_release] != noone)
    {
        script_execute(scripts[ev_left_release]);
    }
}
else
{
    image_index = 0;
}
```

The Draw GUI event

The Draw GUI event (**Add Event** | **Draw** | **Draw GUI**) will draw both the assigned sprite and anything called using a draw method. In this event, the text will be aligned horizontally and vertically; then, the center button will be used to place the text using its current size and origin. The following is the code for this event:

```
// Set the horizontal and vertical alignment to the
center.
draw_set_halign(fa_center);
draw_set_valign(fa_middle);

// Find the x and y middle of the button.
var mid_x, mid_y;
mid_x = x + image_xscale * (sprite_width  0.5 -
sprite_xoffset);
mid_y = y + image_yscale  (sprite_height * 0.5 -
sprite_yoffset);

// Draw the button's text at this middle point.
draw_text(mid_x, mid_y, str_text);
```

The first two functions executed—`draw_set_halign` and `draw_set_valign`—tell the `draw` functionality how to align text horizontally and vertically; `fa_center` and `fa_middle` are two different constants used to set up the alignment respectively.

Two variables local to this event, `mid_x` and `mid_y`, are then defined. These are calculated by using the position of the button. Then, the size of the sprite is derived and divided in half, subtracting the offset from the obtained sizes. This is all multiplied by the respective scaling so that the origin shifts in proportion with the changes.

Finally, the text is drawn using `mid_x`, `mid_y`, and `str_text`, that is, the text initialized in the **Create** event of the button.

Note

Functions such as `draw_text` will only execute when run in **Draw** and **Draw GUI** events. More about this will be explained later in the chapter.

Scripts – scr_create_button

Now that the button's events have been coded, two scripts will be written. In the first script, a script resource named `scr_create_button` will be used to create buttons at runtime and provide simple information about them, such as their position and displayed text.

scr_random_position

The `scr_random_position` script will be used as an example action for the button. It will randomly place an instanced object within the bounds of the room it is in.

Creating buttons using scripts

If this newly scripted object resource is dragged-and-dropped into the room, the resource should change frames upon the mouse entering and pressing the object; however, no scripts have been assigned, which makes this button rather useless. Now, a script will be written that will create and place buttons as well as assign the different scripts that are tested for and called during the button's different events.

For this script, the previously created script resource `scr_create_button` will be used. Enter the following code in the script:

```
// Create a local variable which will represent the ID
of the instanced button.
var obj_new_button;

// argument0 is the x position, argument 1 is the y
position.
obj_new_button = instance_create(argument0, argument1,
obj_button);
```

```
// argument2 will be the displayed text.
obj_new_button.str_text = argument2;

// argument3 through argument6 will be the four script
IDs.
obj_new_button.scripts[ev_left_button] = argument3;
obj_new_button.scripts[ev_left_press] = argument4;
obj_new_button.scripts[ev_left_release] = argument5;
obj_new_button.scripts[ev_mouse_enter] = argument6;

// The newly created button instance is returned.
return obj_new_button;
```

In this script, a button was created at a specific location, assigned its displayed text, and assigned its four scripts. The following values are those described in that script:

- The `x` value
- The `y` value
- The displayed text
- The `left_button` script ID
- The `left_press` script ID
- The `left_release` script ID
- The `mouse_enter` script ID

Scripting room creation code

Now that a script for instantiating buttons has been created, buttons can actually be created! Another object that executes `scr_create_button` during its **Create** event could be created, but to eliminate the need for a new object resource, a single line of code will be added to the room's creation code. Again, a room's creation code can be accessed by navigating to **settings** | **creation code** from the **Room Properties** window. In this script window, `scr_create_button` will be executed using the following code:

```
// Creates a button centered in the room that will
move to a random position when clicked.
script_execute(scr_create_button, room_width * 0.5,
room_height * 0.5, "1st Button", noone, noone,
```

```
scr_random_position, noone);
```

Again, the previous function simply calls `scr_create_button` and supplies the seven required arguments. Though the third script ID argument is assigned to `scr_random_position`, the button is still pretty inactive since the script hasn't been created yet. This will be done next!

Creating scr_random_position

The `scr_random_position` script is mostly going to be used as an example, but it is still useful and can be applied to a variety of game types. As shown in the following code, this script will first determine the minimum and maximum positions that an object can be placed at while staying within the bounds of a room:

```
/* Declares the x and y ranges for places to stay
within the bounds of the room. /
var x_min, y_min, x_max, y_max;

x_min = sprite_xoffset  image_xscale;
y_min = sprite_yoffset  image_yscale;

x_max = room_width - (image_xscale  (sprite_width -
sprite_xoffset));
y_max = room_height - (image_yscale  (sprite_height -
sprite_yoffset));

// Set the x and y to random position within the
range.
x = random_range(x_min, x_max);
y = random_range(y_min, y_max);

/ If the object is within the bounds of the mouse, the
Mouse Enter Event is triggered; otherwise, the Mouse
Leave Event is triggered. */
if (position_meeting(mouse_x, mouse_y, self))
{
    event_perform(ev_mouse, ev_mouse_enter);
}
else
{
    event_perform(ev_mouse, ev_mouse_leave);
}
```

In the first portion of the code, four local variables were defined, representing the ranges of `x` and `y` without allowing the object to leave the bounds of the room. The minimum of the range is derived by using the `x` and `y` sprite offsets. Then, the maximum of the range is calculated by subtracting the difference of the sprite size and the sprite offset, which is then multiplied by its scale, from the size of the room, as shown in the following snippet:

```
var x_min, y_min, x_max, y_max;

x_min = sprite_xoffset * image_xscale;
y_min = sprite_yoffset  image_yscale;

x_max = room_width - (image_xscale  (sprite_width  -
sprite_xoffset));
y_max = room_height - (image_yscale * (sprite_height –
sprite_yoffset));
```

To determine a position, the supplied built-in function `random_range` is used, supplying the desired minimum and maximum values respectively, as shown in the following code snippet:

```
x = random_range(x_min, x_max);
y = random_range(y_min, y_max);
```

The final section, as shown in the following snippet, mostly deals with the button, but can be applied to any object. Once the object is placed, the built-in function `position_meeting` is used, which tests if the specified `x` and `y` coordinates are contained within the bounds of the supplied object. In this case, the keyword `self` is supplied, which represents the ID of the object instance executing this script. If the mouse positions meet, the object's **Mouse Enter** event will be executed; otherwise, the object's **Mouse Leave** event will be executed. The reason this script can be used by other objects is that even if they lack a **Mouse Enter** or **Mouse Leave** event, the script will still execute safely and not throw a runtime error. The following code shows the final section of the script:

```
if (position_meeting(mouse_x, mouse_y, self))
```

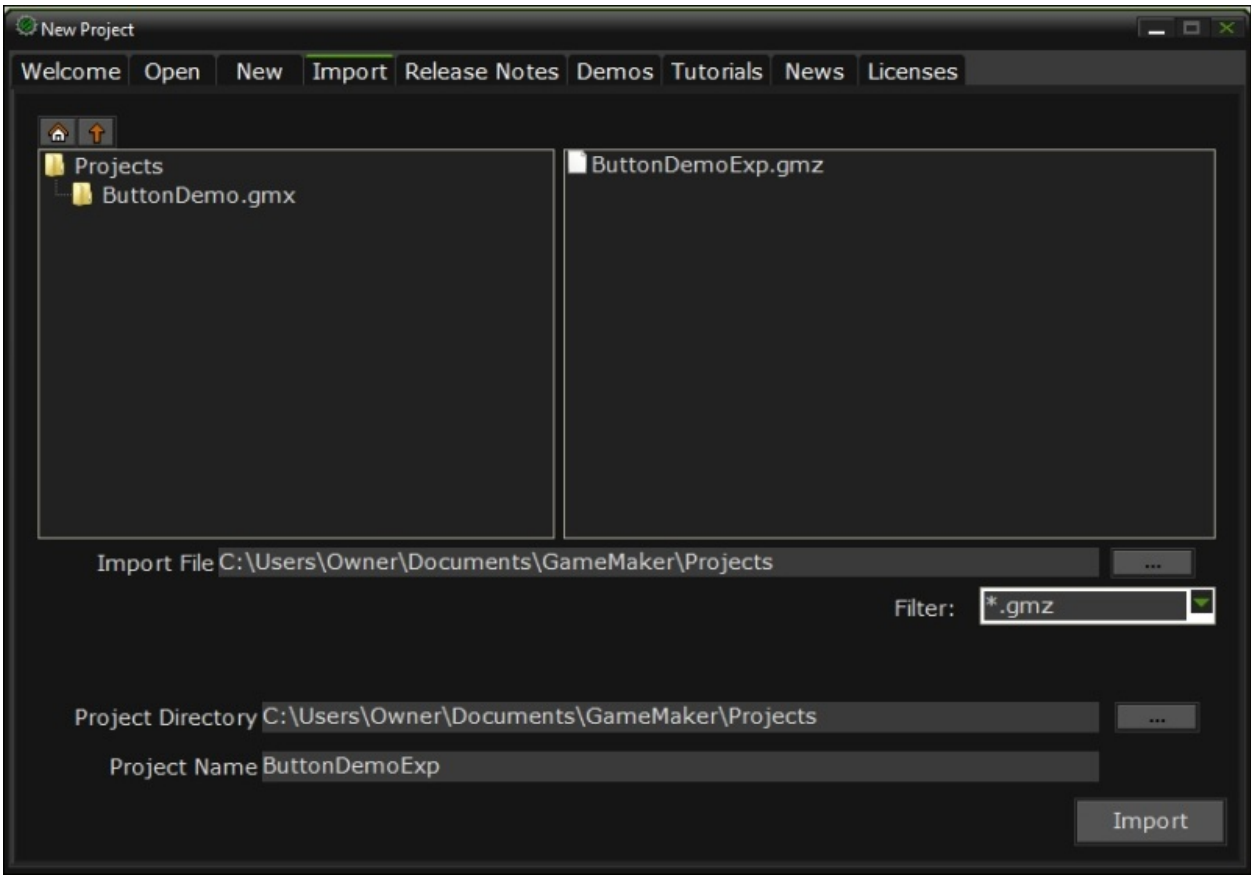
```
{
    event_perform(ev_mouse, ev_mouse_enter);
}
else
{
    event_perform(ev_mouse, ev_mouse_leave);
}
```

Now that the scripts have been properly set up, this game can be run. A button should be visible and centered toward the middle of the room. When the mouse hovers over it, it should go to the second frame and upon being pressed, go to the third. If the mouse is released while still hovering over the button, the button should move to a random spot and its bounds should remain in the room, signifying that `scr_random_position` has been called. If this works and looks similar to the next image, then the creation of this introductory button has been a success!



Exporting and importing the button

Now that the button has been created, this GameMaker project can be exported and then imported into a new one so that it can be re-used without having to copy and paste all of the assets while preserving the original. To do this, go to **File** | **Export Project** and save the compressed GameMaker file. The following screenshot shows the import of our game file:



After creating a new project, an entire project can be imported using **File | Import Project** or by pressing *Ctrl + I* to open a dialog window to import the previously created project. Be warned: this will overwrite all current assets in a project, so it is best used at the beginning of a project. If resources from multiple projects need to be added, this can be done by right-clicking on each asset type in the resource tree and selecting **Add Existing...**. Then, search for the appropriate `.gmx` files for the type of asset being imported, such as `sprite.gmx` for sprites, `object.gmx` for objects, and so on. Unlike importing an entire project, if an asset of the same name already exists, the imported asset will be renamed ending with `_new`.

Summary

In this chapter, the basic syntax and components of GameMaker Language—programs, functions, variables, and so on—were explained to help you start building a strong foundation. A basic button was also created that utilizes many of these concepts. Code was also written so that this modular, easily customized button can be instantiated with just one line of code.

In the next chapter, the first game described in this text will be started: a simple, match-three puzzle game. The foundation for GML scripting, which we started learning about in this chapter, will be built by utilizing and going more in depth with already introduced topics such as built-in variables, for example, `image_xscale`. We will not only focus on placing objects in an organized fashion by using arrays to set up the puzzle game's grid, but also randomization, so that the layout of pieces is seemingly random and unique with each play.

Chapter 2. Random Organization – Creating a Puzzle Game

Now that the basics of GML scripting have been explained, it's time to reinforce those lessons and start developing a game! The game that will be created in this chapter and elaborated upon in a few of the following ones is a puzzle game. The puzzle genre is one comprised of many different types of games; the one that will be focused on is a simple, match-three puzzle game.

Most match-three puzzle games involve a grid of pieces; these pieces are usually differentiated based on some visual indicators, such as color, shape, and so on. Upon swapping two pieces, if either piece completes a row or column of three or more matching pieces, the matching sets are removed and any pieces above fall into the newly emptied positions with new ones subsequently occupying the positions of those that just fell.

In this chapter, the groundwork needed to start creating the puzzle game will be demonstrated. This groundwork includes the following:

- Placing instanced objects within a grid
- Storing information about the grid in a two-dimensional array
- Randomizing the placed pieces

Understanding sprite variables and functions

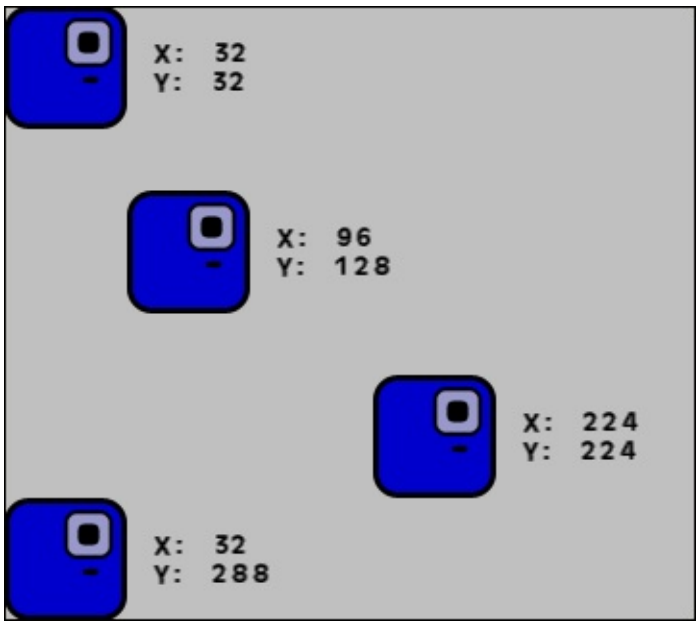
Before starting the puzzle game, it would be a good idea to discuss the different variables and functions associated with the placement and visual manipulation of objects. Because of the geometric layout of a puzzle game, drag-and-drop object placement can be tedious; for example, a 10 X 10 grid requires 100 objects to be laid out; instead, objects will be placed dynamically. To make the drag-and-drop object

placement happen successfully, however, the objects' sizes and origins must be obtained. Fortunately, GML provides many different variables and functions that can make obtaining these values, and thus the proper placement of these objects, easier.

Positioning and origin

Two important built-in, instanced variables dealing with position are `x` and `y`. The `x` instance of an object refers to its horizontal position; the `y` instance of an object refers to its vertical position.

The origins or offsets of an object are actually defined in the **Sprite Properties** window. The values `sprite_xoffset` and `sprite_yoffset`, which deal with the horizontal and vertical origins of a sprite respectively, are instanced constants and cannot be assigned through code like `x` and `y`. The following screenshot shows several sprites, all of which have their origins centered, at different locations:



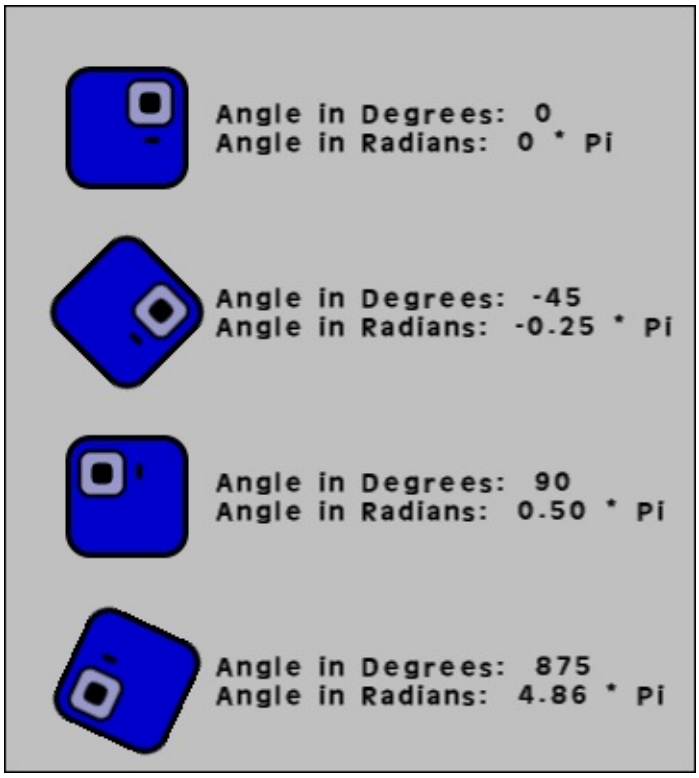
Layering with depth

There is then the matter of deciding the order in which objects will be drawn or the value for `depth`, which by default is set to `0`. Objects with lower values for `depth` are drawn on top of those with higher values as if

they are on different layers. For example, an object with a `depth` value of `-10` will be drawn on top of one with a `depth` value of `10`. The `depth` value only affects the order in which objects are drawn onscreen, not the way they interact with one another; two objects with different depths can still collide with one another.

Rotating

The visual rotation of an object can be handled using `image_angle`. This value must be in **degrees** as opposed to **radians**. Degrees represent the angle and direction of an object's rotation, ranging from 0 to 360 for one full rotation. Radians, on the other hand, are the actual values used and derived from trigonometric functions such as sine, cosine, tangent, and so on. Radians range from 0 to 2π for one full rotation. (In GML, π is represented by the built-in constant `pi`.) To convert a radian value to a degree value for variables such as `image_angle`, the built-in function `radtodeg` can be utilized. If you want to use a degree value in a trigonometric function, it can be converted from degrees to radians using `degtorad`. The next screenshot showcases the earlier sprite rotated at different angles and the accompanying degree and radian values:

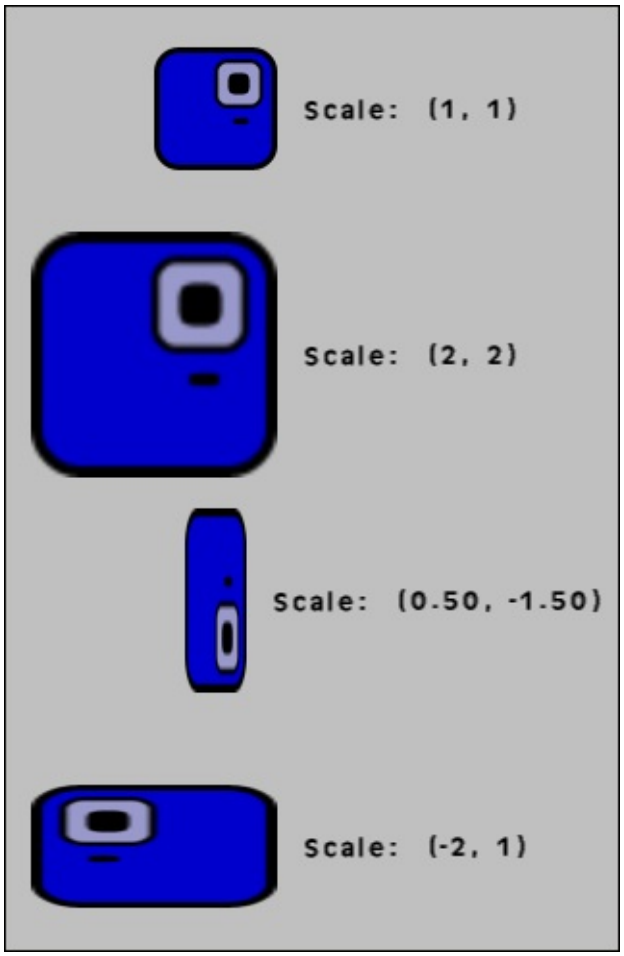

Angle in Degrees: 0
Angle in Radians: 0 * Pi
Angle in Degrees: -45
Angle in Radians: -0.25 * Pi
Angle in Degrees: 90
Angle in Radians: 0.50 * Pi
Angle in Degrees: 875
Angle in Radians: 4.86 * Pi

Note

Objects are rotated around their sprites' offset points. For example, if an object's `x` and `y` offsets are `0`, the object will rotate around its upper-left corner.

Scaling

Objects can also be scaled using the built-in variables `image_xscale` and `image_yscale`. Scale is a multiplicative percentage that defaults to the value `1`. If a negative scale is used, it will give objects the appearance of being flipped or mirrored. Similar to rotation, the point at which an object is scaled horizontally or vertically is determined by the offset of its sprite. When the values of `image_xscale` and `image_yscale` are equal, it is known as **uniform scaling**; when they are not the same, it is known as **nonuniform scaling**. The following screenshot demonstrates sprites being affected by both uniform and nonuniform scaling:



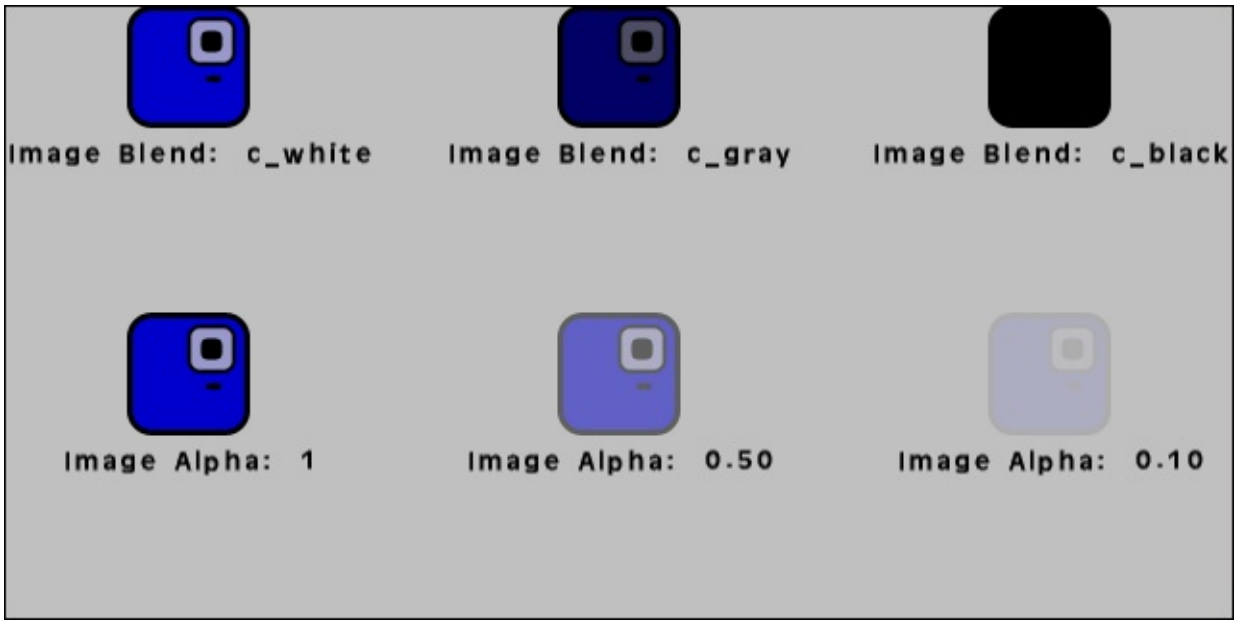
Working with frames and color

In an object, `sprite_index` refers to the ID of the sprite resource being used to draw the object; meanwhile, `image_index` refers to the frame or subimage within the sprite that is displayed. For example, the button created in the previous chapter had three subimages. If `image_index` is set to a value that is out of bounds, for example, `25`, while the sprite only has `20` subimages, the subimage at the `image_index` value that is `5` will be drawn. The `image_index` ID can also represent a fractional value such as `1.25`; in this case, the drawn subimage is the index of the rounded-down value.

All objects loop through these indexes using a variable called `image_speed`. This value is dependent on the speed of the current room. Most room speeds default to `30`, meaning that an object with an

`image_speed` value of `1` would animate at a rate of 30 frames a second. Assigning a value greater than `1` will make the images cycle at a faster rate; a value between `0` and `1` will make the images cycle slower. An `image_index` value of `0` will stop the images from cycling, and a negative value will make them cycle in reverse.

The transparency and blended color of any object can also be changed using the built-in instanced variables `image_alpha` and `image_blend`. The `image_alpha` variable requires a value ranging from `0` (completely invisible) to `1` (fully opaque). This value is multiplicative in that if a pixel in the sprite is already transparent, it will not become opaque if the `image_alpha` value is assigned to `1`. The `image_blend` variable uses a color and multiplies or blends all the pixels in the sprite with that color. By default, the value of `image_blend` is equal to the constant value of `c_white`. GameMaker: Studio provides many other colors as constants, such as `c_red` and `c_blue`; however, color values can also be created using the built-in function `make_color_rgb`, which returns a color value after supplying the `red`, `green`, and `blue` values. There is also `make_color_hsv`, which returns a color value after supplying `hue`, `saturation`, and `value` (or lightness) of the desired color. All supplied arguments are modified to values greater than or equal to `0` and less than `256`. The following screenshot demonstrates the effects of setting `image_blend` and `image_alpha` to different values:



Note

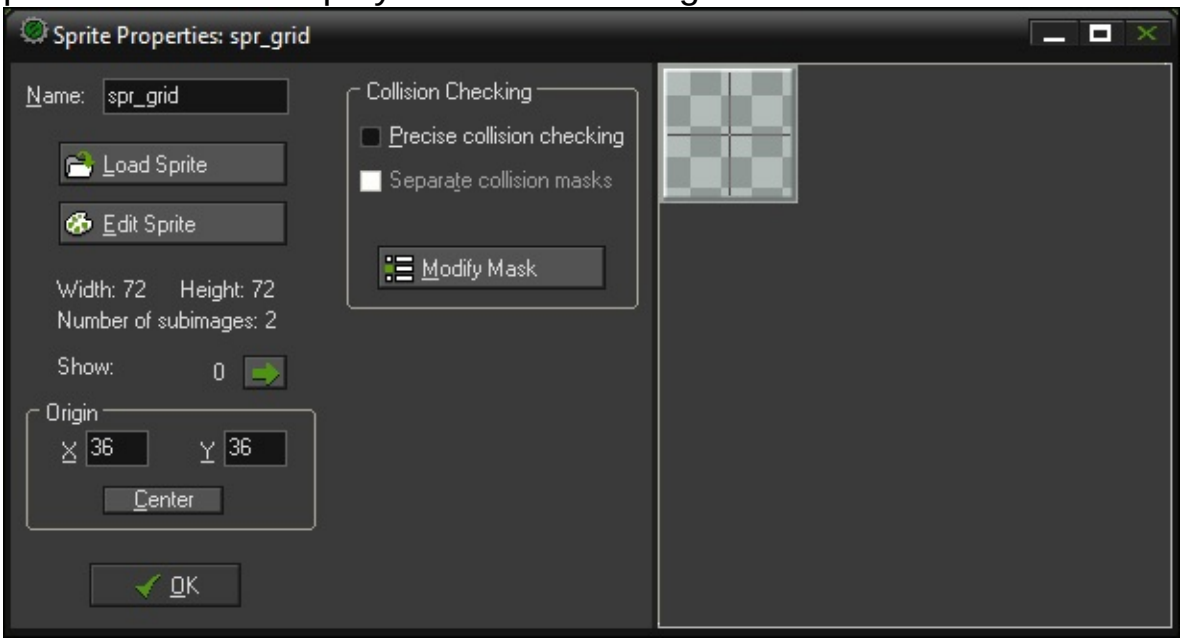
It should be noted that `image_alpha` and `image_blend` have been reported as being problematic on newer platforms such as HTML5. GameMaker: Studio has recently added the ability to work with shaders, but this is a rather advanced topic. More on shaders can be found in GameMaker: Studio by navigating to **Help** | **Contents...**.

Setting up the puzzle game

Because of the simplicity of creating this puzzle game, there are not a lot of sprite resources needed. You will only need two: one representing an empty grid and another that displays the puzzle pieces. There will, however, also be accompanying object, room, and script resources created; these will be described in the upcoming sections.

Sprites – spr_grid

The `spr_grid` sprite represents a single grid block where a piece of the puzzle will be placed. It has only one subimage for now. Load this sprite by clicking on **Load Sprite** and navigating to **PuzzleGame00** | **sprites** | **Images** | **spr_grid_0** from the code folder for this chapter. This sprite is 72 pixels square with a centered origin at 36, both horizontally and vertically. The entire bounding box should be used as well. Its parameters are displayed in the following screenshot:

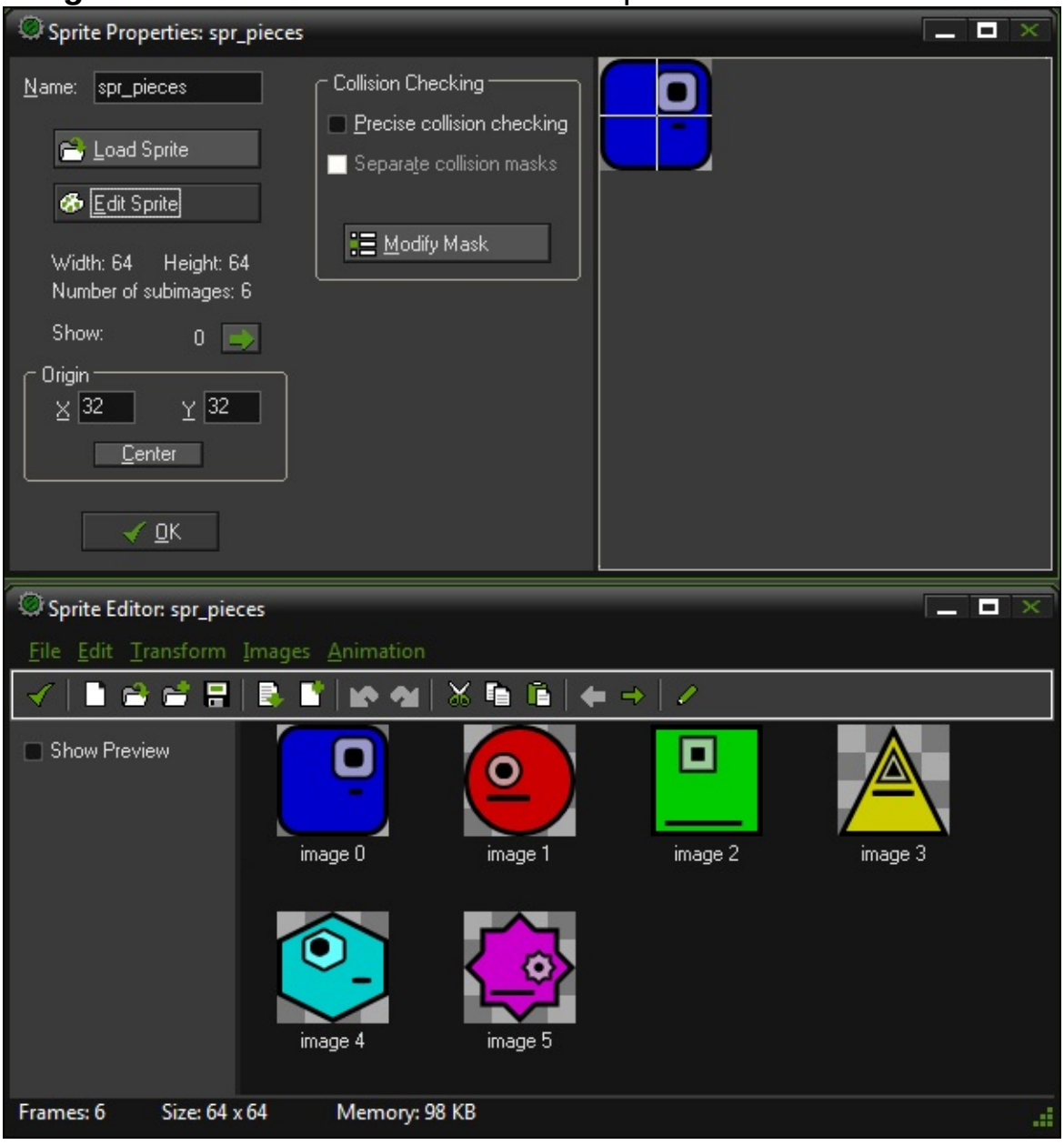


spr_pieces

The `spr_pieces` sprites will represent pieces displayed on the game board. There are six frames in total, each of which differs in color and

shape to make finding matches in the grid easier for players, especially those with color blindness.

This sprite resource is a 64 pixels square with its origin centered at 32 both horizontally and vertically. Its bounding box uses the entire collision area. The following screenshot shows the parameters for the **Sprite Properties** window at the top half of the screen and the sprite frames themselves at the bottom half. These parameters can be loaded by clicking on **Load Sprite** and navigating to **PuzzleGame00** | **sprites** | **Images** from the code folder for this chapter:



Objects – obj_grid_manager

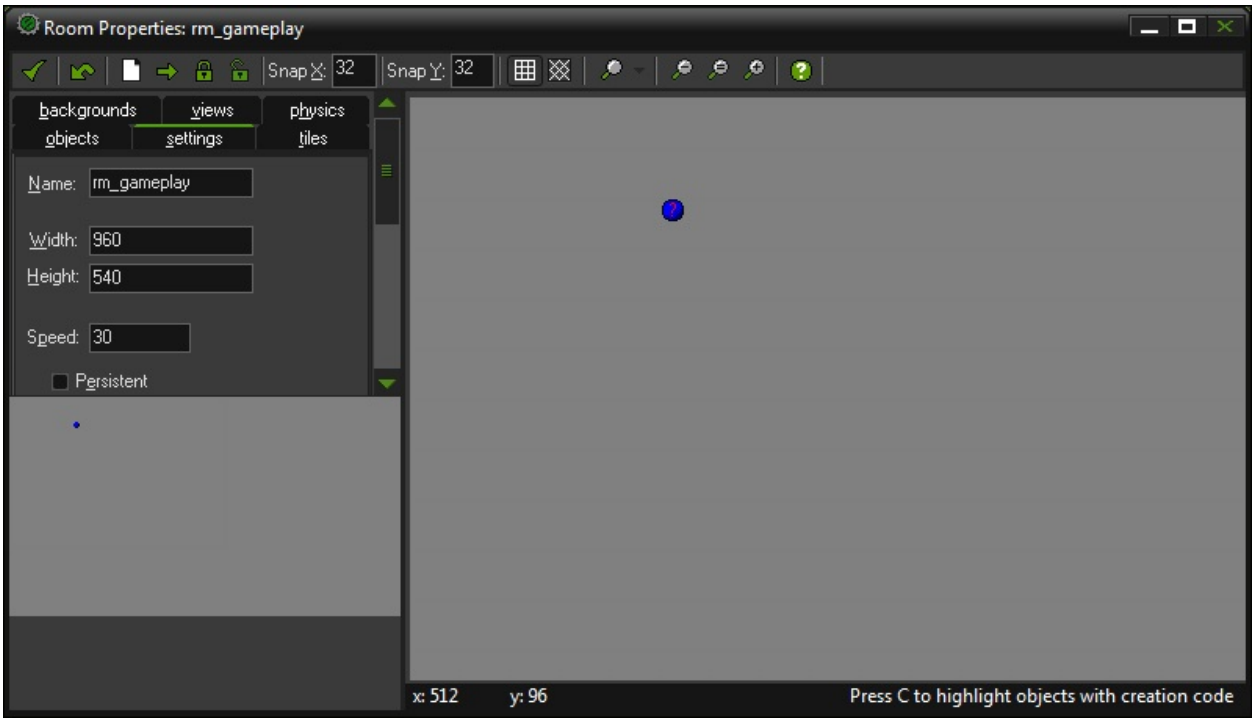
For this project, three object resources will need to be created. The first `obj_grid_manager` object resource will manage the creation and positioning of the puzzle pieces. This object will also have a **Create** event.

obj_grid_block and obj_puzzle_piece

The `obj_grid_block` and `obj_puzzle_piece` objects are very similar. The `obj_grid_block` object will represent the location in which pieces are placed and `obj_puzzle_piece` will represent the actual pieces. The `obj_grid_block` object should be assigned `spr_grid` as its sprite; the `obj_puzzle_piece` object should be assigned `spr_pieces` as its sprite. Neither of these objects should contain any events at this time.

Room – rm_gameplay

The `rm_gameplay` room is where the main gameplay of the puzzle game will take place. This will be resized from the default 640 x 480 to 960 x 540. The room will have only one object placed within it, which is an instance of `obj_grid_manager`. This instance of `obj_grid_manager` should be placed at 160 on the *x* axis and 64 on the *y* axis. This will be the location of the object at the upper-left corner of the grid. Initially, the grid created is going to be 10 columns wide and 6 rows high. This position will ensure that the grid is well centered within the room. The following screenshot shows the placement of `obj_grid_manager`, represented by the blue circle with a red question mark in the otherwise empty room:



Scripts

There will be several script resources created and explained in the following sections of this chapter:

- `scr_create_in_grid`: This script instantiates objects into a grid space
- `scr_get_puzzle_piece`: This script gets an object from the game board, safely returning `noone` if out of range of the game board
- `scr_check_adjacent`: This script is used to check the number of matching pieces that are adjacent to a specified starting point
- `scr_check_board`: This script checks the entire board for matches

Together, these scripts will create a grid populated with puzzle pieces. The pieces will then be checked for matches, and sections with matching sets will be continually changed until the player has a fresh board. The details of each will be explained as the puzzle game is constructed.

Aligning the pieces of the puzzle game to a grid

Now that the necessary resources have been created, scripts can be written to lay out the initial pieces of this puzzle game. The first script will be the **Create** event of `obj_grid_manager`. This script will define several variables within `obj_grid_manager`. It will also execute `scr_create_in_grid` to populate its pair of two-dimensional arrays.

The Create event of obj_grid_manager

The **Create** event of `obj_grid_manager` will create the grid in which the objects are placed. The first portion of the code, shown as follows, will define several instanced variables that will be referenced throughout the game:

```
// Defines the number of columns in the grid.
columns = 10;

// Defines the number of rows in the grid.
rows = 6;

// Set the horizontal and vertical spacing based on
the size of
  the grid block.
x_spacing = sprite_get_width(spr_grid);
y_spacing = sprite_get_height(spr_grid);
```

The `columns` and `rows` variables represent the number of columns and rows that will be used in the game. In this example, there will be 10 columns and 6 rows, making a total of 60 pieces. These numbers can be changed to create a variety of different grids and boards but should be dealt with carefully. For example, a board with 100 columns and 100 rows would not only be difficult to render, but also the time required to check matches would be very long. Simply put, the more objects and processes the game has to work with, the more memory and time they

will take to execute.

Then, the horizontal and vertical spacing between pieces is defined. Since the ID of the sprite that represents the grid is known, its width and height can be stored as instanced variables. These are stored as instanced variables since their values will be used as the pieces are shifted around.

After defining the spacing, two `for` statements—one within the other—are executed. The use of two `for` statements like this is a very important, useful, and common way to iterate through two-dimensional arrays. The following code shows the loop through the columns and rows, creating the grid and puzzle pieces:

```
// Loop through the columns and rows, creating the
grid and puzzle pieces.
for (i = 0; i < columns; i++)
{
  for (j = 0; j < rows; j++)
  {
    array_grid[i, j] = scr_create_in_grid(x, y,
x_spacing,
      y_spacing, 1, obj_grid_block, i, j);
    array_pieces[i, j] = scr_create_in_grid(x, y,
x_spacing,
      y_spacing, 0, obj_puzzle_piece, i, j);
  }
}
```

In the previous code, objects representing grid blocks and individual pieces are created and placed using the script `scr_create_in_grid`, which will be explained in the next section. The utilized script returns a value, which is the ID of the newly created object, and stores the grid and the pieces in their respective two-dimensional arrays.

The following is the current **Create** event script in its entirety:

```
/// Creates the grid and the pieces on the board,
storing them into arrays.

// Defines the number of columns in the grid.
```

```
columns = 10;

// Defines the number of rows in the grid.
rows = 6;

// Set the horizontal and vertical spacing based on
the size of the grid block.
x_spacing = sprite_get_width(spr_grid);
y_spacing = sprite_get_height(spr_grid);

// Define variables for loops.
var i, j;

// Loop through the columns and rows, creating the
grid and puzzle pieces.
for (i = 0; i < columns; i++)
{
  for (j = 0; j < rows; j++)
  {
    array_grid[i, j] = scr_create_in_grid(x, y,
x_spacing,
      y_spacing, 1, obj_grid_block, i, j);
    array_pieces[i, j] = scr_create_in_grid(x, y,
x_spacing,
      y_spacing, 0, obj_puzzle_piece, i, j);
  }
}
```

scr_create_in_grid

Now that the **Create** event of `obj_grid_manager` has been written, the script it references—`scr_create_in_grid`—can be coded. This script handles the instantiation, initial variable assignment, and placement of objects. It is a useful script that allows the populating of a room in an organized fashion. Also, the ID of the new object is returned so it can be used or stored in the `obj_grid_manager` object's pair of two-dimensional arrays. The following is the code for this script:

```
/// Creates a new instance adhered to a grid.

/* A comment block used to keep track of all the
arguments in this function.
argument0 -- the grid's starting x position,
argument1 -- the grid's starting y position,
```

```
argument2 -- the horizontal spacing between objects in
the grid
argument3 -- the vertical spacing between objects in
the grid.
argument4 -- the new object's depth
argument5 -- the object index of the to be created
instance.
argument6 -- the column that the object will be
placed.
argument7 -- the row that the object will be placed.
/

var new_instance, x_pos, y_pos;

// The horizontal and vertical placement of the sprite
is determined.
x_pos = argument0 + argument2  argument6;
y_pos = argument1 + argument3 * argument7;

// The new instance is created
new_instance = instance_create(x_pos, y_pos,
argument5);

// The depth of the new instance is set.
new_instance.depth = argument4;

// The image_speed is set to 0 on the new instance.
new_instance.image_speed = 0;

// Two instanced variables are set to the new object
representing its current column and row
new_instance.col = argument6;
new_instance.row = argument7;

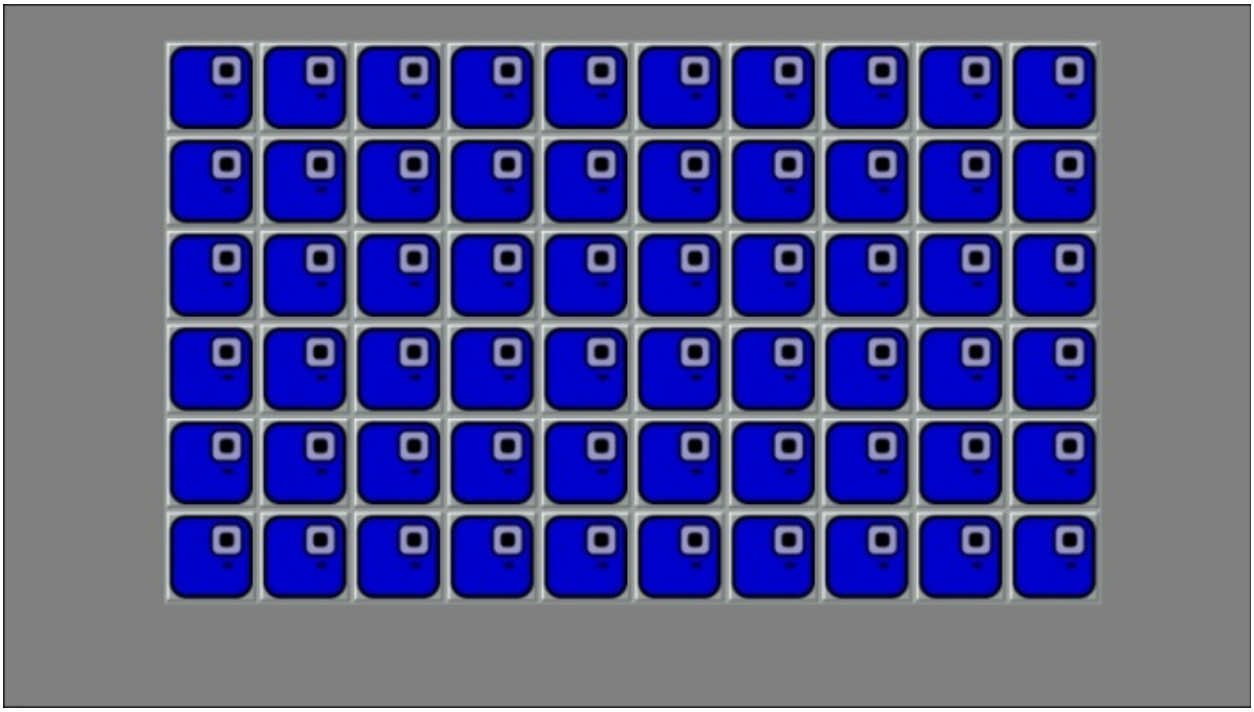
// The new instance is returned.
return new_instance;
```

In the previous code, the following happens:

- The horizontal and vertical positions are defined by fetching the starting position and adding it to the result of the respective spacing multiplied by the column or row index
- The new instance is then created at the specified location
- Its depth is then set, and its `image_speed` value is set to `0`, so the object doesn't cycle through its subimages

- The instance then has two instanced variables defined: `col` (which represents column) and `row` (which will be used for placing and tracking the object)
- Finally, the new instance is returned

Now that `scr_create_in_grid` has been created, the game should be able to be run. If everything was set properly, the game should look something like the following:



Now, of course, this doesn't look like much of a puzzle game. Though the pieces are nicely organized, they are all the same, which would cause one large match on startup! Now it's time to add some randomization to the board.

Understanding and utilizing randomization

A lot of different games use randomization to create a sense of unpredictability, serving as a challenge to players. In the case of this puzzle game, randomness refers to the selection of each piece type in the grid. Contrary to what is observed, the randomness in most games isn't truly random; instead it is a nearly unpredictable pattern of numbers.

In GML, this pattern is determined using an integer value called `seed`. This `seed` value can be obtained using the built-in function `random_get_seed`; it can also be set using `random_set_seed`. Storing or setting the value of the randomization `seed` can be useful when debugging the initial setup of a game so that a particular pattern is always the same. When a game is finally released, however, it is important to call the `randomize` function, which will assign a random seed to be used in the game; otherwise, the game will always use the same random seed by default.

Random functions

The following code sample demonstrates all of the different random functions and their details:

```
// Returns a value between 0 and the given argument.
// 0 is inclusive, meaning the returned value could be
0.
// The argument is exclusive meaning the returned
value will// always be less.
a = random(12.4);

// Returns a value between the first and second
argument.
// The first argument is inclusive; the second
argument is
// exclusive.
b = random_range(23, 192);
```

```
// Returns a random integer between 0 and the given
argument.
// In the case of irandom, the argument is inclusive.
c = irandom(32);

// Returns a random integer between the two supplied,
inclusive
// values.
d = irandom_range(43, 993);

// Returns one of the 16 or less provided arguments.
// The arguments can be string, reals, object ids,
etc.
e = choose("one", "two", "three", "four", "five");
```

Randomizing the board

Now that the basics of randomness have been introduced, they can be applied to the game. Going back to the **Create** event of `obj_grid_manager`, the `irandom` value can be used to assign the pieces to a random subimage. This will be done with three additions to this event.

This first addition is a quick call to `randomize` at the beginning of the code, as shown in the following code:

```
// Ensures random functions produce unique results
with each play.
randomize();
```

The previous line of code will ensure that `irandom` and similar functions are actually randomly used and should be the first line of code. Again, this can be temporarily commented out while debugging or just starting out.

Then a variable is defined that will represent the range of pieces used in the game, as shown in the following code:

```
// The number of pieces that will be present on the
board.  Should
// be greater than 0.
```

```
piece_range = 3;
```

The previous code should be placed near the beginning of the code where variables such as the number of columns and rows are defined. The `piece_range` value represents the range of different pieces that will be used in the grid. In this example, up to six different pieces can be used since `spr_pieces` has six subimages. This number must be set carefully though. Using `0`, for example, would create a grid with all the pieces of the same color, which would be cleared since the entire board would match; then the next set and so on would also match. Smaller numbers, such as `1` and `2`, will work, but could create a long series of matches, forcing the player to just watch the pieces break over and over again. In this example, `3` will be used, which will use four pieces since `irandom` is inclusive. These four pieces will include those with the subimages `0` through `3` in `spr_pieces`. This is demonstrated in the following code:

```
for (i = 0; i < columns; i++)
{
    for (j = 0; j < rows; j++)
    {
        array_grid[i, j] = scr_create_in_grid(x, y,
x_spacing, y_spacing, 1, obj_grid_block, i, j);
        array_pieces[i, j] = scr_create_in_grid(x, y,
x_spacing, y_spacing, 0, obj_puzzle_piece, i, j);

        // Randomizes the image_index of the piece.
        array_pieces[i, j].image_index =
irandom(piece_range);
    }
}
```

After the pieces are instantiated, a line has now been added that sets the `image_index` value of the newly created objects to a value ranging from `0` to `piece_range`. The following is the current state of the `obj_grid_manager` object's **Create** event in its entirety:

```
/// Creates the grid and the pieces on the board,
storing them into arrays.
```

```
// Ensures random functions produce unique results
with each play.
randomize();

// Defines the number of columns in the grid.
columns = 10;

// Defines the number of rows in the grid.
rows = 6;

// The number of pieces that will be present on the
board Should
// be greater than 1.
piece_range = 3;

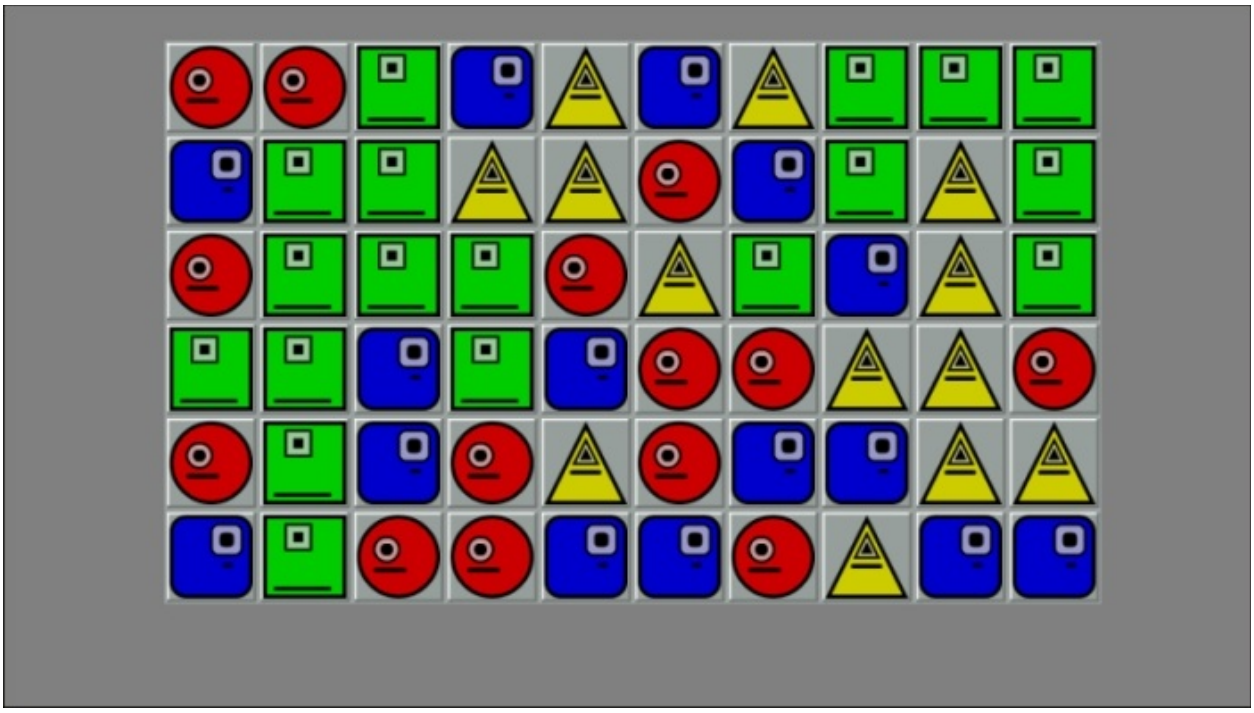
// Set the horizontal and vertical spacing based on
the size of
// the grid block.
x_spacing = sprite_get_width(spr_grid);
y_spacing = sprite_get_height(spr_grid);

// Define variables for loops.
var i, j;

// Loop through the columns and rows, creating the
grid and puzzle // pieces.
for (i = 0; i < columns; i++)
{
    for (j = 0; j < rows; j++)
    {
        array_grid[i, j] = scr_create_in_grid(x, y,
x_spacing, y_spacing, 1, obj_grid_block, i, j);
        array_pieces[i, j] = scr_create_in_grid(x, y,
x_spacing, y_spacing, 0, obj_puzzle_piece, i, j);

    // Randomizes the image_index of the piece.
    array_pieces[i, j].image_index =
irandom(piece_range);
    }
}
```

If these new code fragments have been added to the game properly, the game board should look something like the following:



In the previous screenshot, observe that some pieces are already making matching groups of three! In the next section, scripts will be created that will recursively check for matching groups and change them so that the player can start with a fresh board.

Checking pieces

The board is now built with a random configuration; however, there may already be sets of matching pieces. Of course, just because a player can see a set, does not mean the game can.

To find matching sets of three or more pieces on the game board, the game will start at the upper-left corner of the screen. It will then check the pieces below and to the right of this initial piece. If there are three or more matching pieces in the same direction, they will be re-randomized. The board will then continue to be checked. If there are sets of matching pieces, the entire process is repeated. This is known as a **recursive** process due to its repetitive nature. It will continue to be executed until its goal of starting the player out with a fresh board, where there are no adjacent sets of three or more matching pieces, is achieved.

scr_get_puzzle_piece

The first thing the game should be able to do is return any piece from the board; however, due to the recursive nature of the function that will be used to check for pieces on the board, a requested column or row may be out of range. Because of this, it's smart to create a script to safely reference objects from the `obj_grid_manager` object's two-dimensional array `array_pieces`, as shown in the following code:

```
// Gets an item from a two-dimensional array while
staying in range.

/*
argument0 -- the column index
argument1 -- the row index
*/

if (argument0 >= 0 && argument0 < columns && argument1
>= 0 && argument1 < rows)
{
    // Returns the object from the 2D array.
    return array_pieces[argument0, argument1]
```

```
}
else
{
    // Returns noone if the arguments are out of
range.
    return noone;
}
```

This script can only be used by instances of `obj_grid_manager` since `column`, `row`, and `array_pieces` are all instanced variables. Using a set of Booleans, the first and second arguments are checked to see whether they are greater than or equal to `0` and then less than the number of columns and rows respectively. If the first or second arguments are out of range, the built-in constant `noone` is returned, which will represent a nonexistent object.

scr_check_adjacent

The following script uses recursion to obtain the number of pieces adjacent to a specified starting piece at the given column and row. It then increases this value, keeping track of the value until a piece that doesn't match is encountered or a column or row that is out of range is requested. The first part of this script, as shown in the following code, will establish local variables and then get the adjacent piece:

```
/// Finds the total number of matching, adjacent
pieces

/*
argument0 the column to check.
argument1 the row to check.
argument2 The direction in which to check pieces
horizontally; should be equal to -1, 0, or 1.
argument3 The direction in which to check pieces
vertically; should be equal to -1, 0, or 1.
*/

// Declares variables.
var current_piece, col, row, other_piece, count;

// Assigns the local column and row variables.
```

```
col = argument0;
row = argument1;

// Tracks the number of adjacent pieces with the same
image_index.
count = 0;

// Gets the current piece.
current_piece = scr_get_puzzle_piece(col, row);

// If a null piece is returned, the function returns
0.
// If the column and row are both 0, the method ends
to avoid an infinite loop.
if (current_piece == noone || (argument2 == 0 &&
argument3 == 0))
{
    return 0;
}

// col and row are adjusted based on the provided
arguments.
col += argument2;
row += argument3;

// Gets the checked piece.
other_piece = scr_get_puzzle_piece(col, row);
```

This script requires four arguments. The first two refer to the column and row indices of the current piece being tested. Then, `argument2` refers to the horizontal offset while `argument3` refers to the vertical offset. It's commented in the preceding code that both of these should be either `-1`, `0`, or `1`. Though other numbers could work, the goal of this function is to check adjacent pieces. Five local variables are then created as shown in the following list:

- `col`: This variable represents columns and is used to avoid having to type `argument0` several times
- `row`: This variable represents the row index and is used to avoid having to type `argument1` multiple times
- `count`: This variable gives the total number of adjacent pieces
- `current_piece`: This variable gives the instance ID of the currently checked piece

- `other_piece`: This variable gives the instance ID of the adjacent piece

The `current_piece` variable is then assigned using `scr_get_puzzle_piece`. It is then checked, and if it is unassigned or `argument2` and `argument3` are both `0`, the function returns `0`. Next, `col` and `row` are increased by the respective arguments and `other_piece` is set using `scr_get_puzzle_piece` with the updated values of `col` and `row`.

The second part of this function applies recursion by calling itself once again with the updated values of `col` and `row` and adding the result to the `count` variable, as shown in the following code:

```
// If the other piece is defined and matches the
current pieces image_index, count is increased.
if (other_piece != noone && other_piece.image_index ==
  current_piece.image_index)
{
  // Increase count by one.  The function is called
again with
    the adjusted rows.
  // Continue to check, but from the new column and
row
  count += 1 + scr_check_adjacent(col, row, argument2,
argument3);
}

// Return the final count.
return count;
```

So, once again, `other_piece` is checked to make sure it is not equal to `noone`, which means that it is referring to a piece on the board and if it is, the value for `image_index` is compared to that of `current_piece`. If these are equivalent, the value of `count` is increased by `1` and the result of another execution of `scr_check_adjacent` with the updated values for `col` and `row`. This recursive function will ultimately get the total number of adjacent pieces in the direction specified by `argument2` and `argument3`. The entirety of this function is as follows:

```
/// Finds the total number of matching, adjacent
pieces
```

```
/*
argument0 the column to check.
argument1 the row to check.
argument2 The direction in which to check pieces
horizontally; should be equal to -1, 0, or 1.
argument3 The direction in which to check pieces
vertically; should be equal to -1, 0, or 1.
*/

// Declares variables.
var current_piece, col, row, other_piece, count;

// Assigns the local column and row variables.
col = argument0;
row = argument1;

// Tracks the number of adjacent pieces with the same
image_index.
count = 0;

// Gets the current piece.
current_piece = scr_get_puzzle_piece(col, row);

// If a null piece is returned, the function returns
0.
// If the column and row are both 0, the method ends
to avoid an infinite loop.
if (current_piece == noone || (argument2 == 0 &&
argument3 == 0))
{
    return 0;
}

// col and row are adjusted based on the provided
arguments.
col += argument2;
row += argument3;

// Gets the checked piece.
other_piece = scr_get_puzzle_piece(col, row);

// If the other piece is defined and matches the
current pieces image_index, count is increased.
if (other_piece != noone && other_piece.image_index ==
  current_piece.image_index)
{
```

```
  // Increase count by one.  The function is called
again with
    the adjusted rows.
  // Continue to check, but from the new column and
row
  count += 1 + scr_check_adjacent(col, row, argument2,
argument3);
}

// Return the final count.
return count;
```

This function takes a pair of column and row indexes. The `current_piece` value at this board position is obtained. If it is valid, the column and row indexes are incremented by the respective arguments, and these pieces are pulled from the board. Once this piece is obtained, again, if it is valid, its `image_index` value is checked and compared to that of the starting piece. If their values are equal, the value of `count` is increased by `1` and `scr_check_adjacent` is run again. This recursive method call is done so that the total number of matching pieces is received, stopping when one of the requested indexes is out of range or a returned piece's `image_index` value does not match.

scr_check_board

Now that the scripts that can get pieces from the board and the number of matching pieces from a provided column and row have been specified, the entire board can be checked. This is also a recursive function, whose goal was specified earlier: to continually randomize pieces until there are no matching sets of three or more.

In the first part, local variables are defined, as shown in the following code:

```
/// Checks each piece on the board for matches.

/* argument0 -- if true, designates matching sets of
three or
  more to be destroyed; otherwise, matching sets are
continuously adjusted.
```

```
*/

// Create the loops variables as well as a variable
designating that matches have been found.
var i, j, match_found;

// Set to true so the initial while loop runs at least
once.
match_found = true;
```

The `scr_check_board` function is very important and will also be used in the next chapter, so it does contain some aspects that have not been touched upon yet, particularly the use of `argument0`. For now, the most relevant local variables are `i` and `j`, which will be used in the `for` statements, and `match_found`, a Boolean that indicates whether or not a match has yet been encountered. After initializing the different local variables, `match_found` is set to `true`.

In the next part of the code, a `while` statement is run that checks the value of `match_found`:

```
while (match_found)
{
    // match_found set to false.  If no matches are
found, the
    loop will be broken.
    match_found = false;

    // Loop through each column and row.
    for (i = 0; i < columns; i++)
    {
        for (j = 0; j < rows; j++)
        {
            // The count of adjacent, matching pieces
to the right and below.
            var h_count, v_count;
            h_count = scr_check_adjacent(i, j, 1, 0);
            v_count = scr_check_adjacent(i, j, 0, 1);
```

Because `match_found` was set to `true`, the `while` statement will run at least once. Within the `while` statement, `match_found` is then set to `false`. This is done so that if the next set of checks performed returns no

matches, the `while` statement will end.

Then, a pair of `for` statements is executed. Just as when the board was constructed, this ensures that every piece is checked. Two more local variables are then defined—`h_count` and `v_count`—which keep track of the number of matching horizontal and vertical pieces adjacent to the current piece. These are set by calling `scr_check_adjacent` twice: once with a horizontal offset of `1` and another time with a vertical offset of `1`.

After obtaining the values for `h_count` and `v_count`, the following code is used to randomize these adjacent pieces when the value of either is greater than or equal to `2`:

```
// If the number of match pieces matching to the right
is greater than or equal to 2...
            if (h_count >= 2)
            {
                while (h_count >= 0)
                {
                    array_pieces[i + h_count,
j].image_index = irandom(piece_range);
                    h_count--;
                }
                match_found = true;
            }
```

Focusing on just `h_count` for now, if its value is greater than or equal to `2`, a new `while` statement is run, this one being performed when `h_count` is greater than or equal to `0`. A piece is then pulled from `array_pieces`. This can be done since this function will be called exclusively by an instance of `obj_grid_manager`. The first index is the sum of `i` and `h_count` and the second is the row, represented by `j`. The `scr_get_puzzle_piece` script could be used, but since `scr_check_adjacent` was already executed when assigning the value to `h_count`, this would be redundant. This piece's `image_index` object is then randomized using `irandom` and the value of `h_count` is decremented. After the execution of this `while` statement, `match_found` is set to `true`, which means that matches have been found and that the entire recursive process should be performed once again.

The same is then done for `v_count` (which has been highlighted in the

following code), except that the secondary index of the piece taken from `array_pieces` is the sum of `j` and `v_count`. The following is the entirety of this recursive function:

```
/// Checks each piece on the board for matches.

/* argument0 -- if true, designates matching sets of
three or
    more to be destroyed; otherwise, matching sets are
continuously adjusted.
*/

// Create the loops variables as well as a variable
designating that matches have been found.
var i, j, match_found;

// Set to true so the initial while loop runs at least
once.
match_found = true;

while (match_found)
{
    // match_found set to false.  If no matches are
found, the
      loop will be broken.
    match_found = false;

    // Loop through each column and row.
    for (i = 0; i < columns; i++)
    {
        for (j = 0; j < rows; j++)
        {
            // The count of adjacent, matching pieces
to the right and below.
            var h_count, v_count;
            h_count = scr_check_adjacent(i, j, 1, 0);
            v_count = scr_check_adjacent(i, j, 0, 1);

            // If the number of match pieces matching
to the right is greater than or equal to 2...
            if (h_count >= 2)
            {
                while (h_count >= 0)
                {
                    array_pieces[i + h_count,
j].image_index = irandom(piece_range);
```

```
                    h_count--;
                }
                match_found = true;
            }

            // If the number of match pieces matching
below is greater than or equal to 2...
            if (v_count >= 2)
            {
                while (v_count >= 0)
                {
                    array_pieces[i, j +
v_count].image_index =  irandom(piece_range);
                    v_count--;
                }
                match_found = true;
            }
        }
    }
}
```

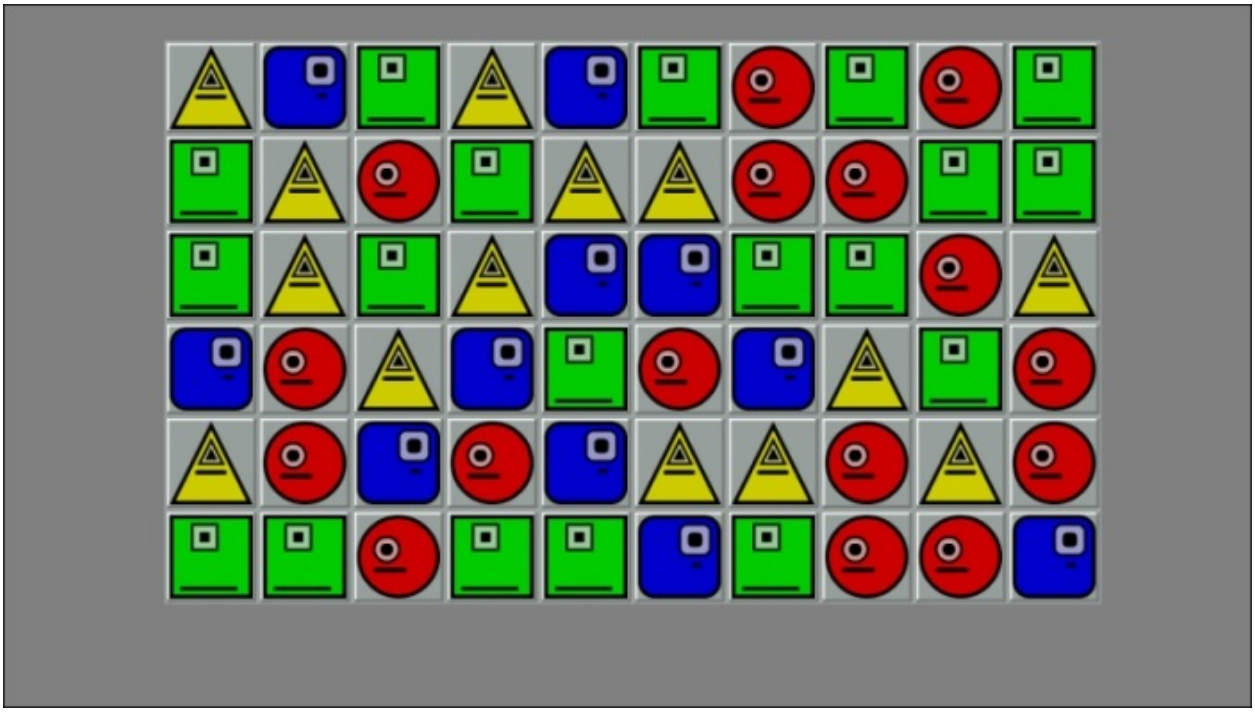
Because the recursion showed in the previous code will most probably be repeated multiple times, there is a chance that as every set of pieces is randomized, the game could continually create new sets for quite some time; however, due to the reduction of the number of matching sets of pieces with each recursion, the chances of this are highly unlikely. Even if `piece_range` is set to `1`, meaning there would only be two piece types, eventually, a board void of matches will be generated.

Running it all together

If the game were to be run now, nothing would be accomplished despite all of the code we've just written. This is because a final call to `scr_check_board` must be made. This should be added to the end of the **Create** event script in `obj_grid_manager` as follows:

```
// Recursively checks the board to prevent matches of
three on
// start.
scr_check_board(false);
```

Now when it's run, the game should generate boards in a grid space with no matching sets—a fresh, clean board for players to enjoy, as shown in the following screenshot:



Summary

This chapter has set up the basic framework for the match-three puzzle game. Puzzle pieces were made and placed on a customizable grid; references to each puzzle piece were stored in a two-dimensional array. Using randomization, the pieces on the board were configured to create a unique pattern. Finally, pieces were recursively checked and compared to one another to look for sets of three horizontally or vertically. If any were found, they were changed so the player could start the level with a fresh board.

The current game is nice to look at, but needs player interaction. In the next chapter, scripts and events will be introduced to add the ability to read mouse and keyboard input. Also, current scripts will be updated so that players can use their ability to interact with and swap pieces on the board in the hope of creating matches!

Chapter 3. So How Do I Play? – Adding Player Interaction

In the previous chapter, a match-three game was started by laying pieces into a grid using a randomized fashion. A recursive function was then utilized so that the player would start with a board free of matching sets. This is nice, but the game is lacking the important element of player interaction.

In this chapter, player input will be focused on, particularly through mouse input and keyboard input. Using these input devices, the player will be given the ability to navigate the board as well as swap pieces to create matching sets. Knowing how to work with and implement multiple input sources can help expand a game's player base. Additional functionality will also be added to existing scripts so that the game reacts when the pieces match.

Designing player interaction

Player interaction is an important aspect of a game's design. It's one element that essentially separates games from other forms of media, such as movies. Designing player interaction entails answering questions like the following:

- How does the player interact with the game?
- What does the player use to interact with the game?
- What does the player see as they interact with the game?

The prior list contains just some of the questions that can be answered when designing player interaction for a game. In this puzzle game, the player will highlight a place on the grid and perform a secondary action to swap pieces on the board either horizontally or vertically using the mouse or keyboard. If the swap creates a matching set of three or more pieces, those pieces will disappear and new pieces will fall into their place. If new matches exist, these too will break, creating a combo. If the swap does

not create a match though, the pieces will return to their original positions.

Reading input with the mouse

Mouse events are those that are triggered through use of the player's mouse. When adding events to an object resource, mouse events can be found under the **Mouse** button in the **Choose the Event to Add** dialogue window.

There are various mouse events, some of which were discussed in Chapter 1, *Getting Started – An Introduction to GML*, but the following list goes through each of them in more detail:

- **Left button**, **Middle button**, and **Right button**: These events are triggered for every frame when the respective mouse button is held down within the collision mask of the object's sprite
- **No button**: This event is triggered when no mouse button is held down within the collision mask
- **Left pressed**, **Middle pressed**, and **Right pressed**: These events are triggered only once when the respective mouse button is held down within the object's collision mask
- **Left released**, **Middle released**, and **Right released**: These events are triggered only once when the respective mouse button is released while the mouse is pointing within the object's collision mask
- **Mouse enter**: This event is triggered when the mouse enters the object's collision mask
- **Mouse leave**: This event is triggered when the mouse leaves the object's collision mask after entering it
- **Mouse wheel up** and **Mouse wheel down**: These events are triggered when the user's mouse wheel is moved up or down
- **Global mouse**: **Global mouse** has the **Button pressed** and **Button released** events for all three mouse buttons; the difference is that these will occur regardless of the mouse's position in relation to the object's collision mask when the mouse pointer is within the game window

Understanding mouse variables and functions

Mouse input information can be derived during events other than **Mouse** events, such as a **Step** event; this can be done through the use of the following built-in variables:

- `mouse_x` and `mouse_y`: These variables retrieve the horizontal and vertical position of the mouse respectively
- `mouse_button`: This variable retrieves the index of the button currently being pressed
- `mouse_lastbutton`: This variable retrieves the index of the mouse button pressed before the most recent one

When using `mouse_button` and `mouse_last button`, one of the following built-in constants will be returned:

- `mb_left`: The left mouse button
- `mb_middle`: The middle mouse button
- `mb_right`: The right mouse button
- `mb_any`: Any mouse button
- `mb_none`: No mouse button

Finally, the following functions can be used to detect mouse input outside of the **Mouse** events, as explained in the following code sample:

```
var result = "No input";

// Checks whether the specified mouse button is down,
returning a boolean.
if (mouse_check_button(mb_left))
{
    result = "Left mouse button down";
}

/* Checks whether the specified mouse button was
pressed this frame, returning a boolean.  Using
mb_any, will return true regardless of which button
was pressed.
```

```
if (mouse_check_button_pressed(mb_any))
{
    result = "A mouse button was pressed";
}

 Checks whether the specified mouse button was
released this frame, returning a boolean. /
if (mouse_check_button_released(mb_right))
{
    result = "Right mouse button released";
}

// Checks whether the mouse wheel was moved down or up
respectively.
if (mouse_wheel_down() || mouse_wheel_up())
{
    result = "Mouse wheel moved";
}

/ Clears input state for the specified button, meaning
it will not generate events until pressed or released
again. */
mouse_clear(mb_left);

return result;
```

Working with touch devices

GameMaker: Studio does have the ability to create games for touch devices such as the iPad with the GameMaker: Studio Master version, but there are no specific touch events that can be assigned to objects. The left mouse button, however, currently represents a single-finger touch, so if this game were to be deployed to one of these devices, no additional work would need to be done to detect the described input properly.

Creating resources to integrate mouse input into the puzzle game

Now that the basic configuration of mouse input have been discussed, it can now be integrated into the puzzle game. This will require changes to some of the scripts written in the previous chapter.

> **Note**
>
> In some of these script updates, variables for keyboard input will also be added. These will be explained shortly after the mouse input is integrated.

Adding new events to obj_grid_manager

Within `obj_grid_manager`, an update to the **Create** event needs to be made. One of the updates will make the grid manager accessible through a global variable. The other update in the **Create** event will track information about which pieces are being highlighted by the mouse and the keyboard.

The Create event

In the **Create** event of `obj_grid_manager`, several new variables must be initialized. The following script, which is part of another **Execute Code** item, creates several new variables that will be used for mouse and keyboard input:

```
/// Initializes mouse and keyboard input specific
variables.

// A reference to the grid manager.
global.grid_manager = id;
```

```
// The current column and row highlighted on the board
by mouse.
current_mouse_col = -1;
current_mouse_row = -1;

// The current piece highlighted on the board by
keyboard.
current_key_col = -1;
current_key_row = -1;
```

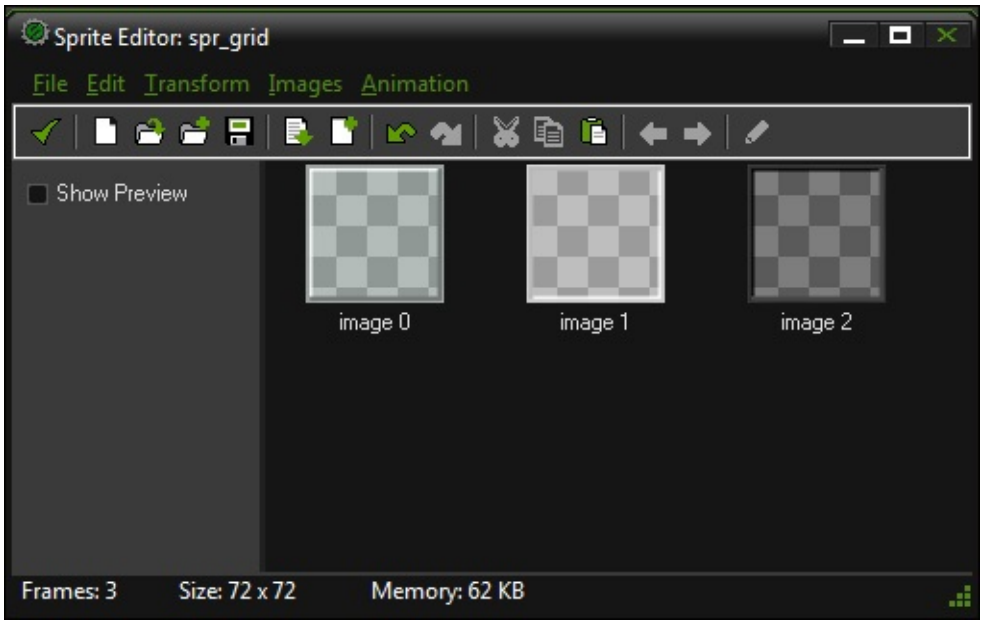
The first global variable, `global.grid_manager`, is assigned the current instance's `id` so that other objects can reference it. The `current_mouse_col` and `current_mouse_row` values represent the column and row currently highlighted using the mouse, and `current_key_col` and `current_key_row` represent the column and row currently highlighted using the keyboard. These are kept separate, so if the player starts using the mouse, the column and row highlighted by the keyboard are deactivated and vice versa.

Updating spr_grid and obj_grid_block

The following updates will be used to indicate that a grid block has been highlighted behind the board pieces. Most of the updates to `obj_grid_block` involve the addition of **Mouse** events.

Adding new frames to spr_grid

Two frames will be added to `spr_grid`; these two additional frames will represent states of the grid block. An `image_index` value of `0` will indicate the idle state, `1` will indicate the highlighted state, and `2` will indicate the swap state: when the grid block is ready to swap the piece in its space with that from an adjacent one. The following screenshot shows the **Sprite Editor** window:



Implementing the Mouse events for obj_grid_block

Now that the sprite for `obj_grid_block` has been updated, it needs the **Mouse** events so that the player can actually interact with them. The following four events will be added:

- **Mouse Enter**
- **Left Pressed**
- **Global Left Released**
- **Mouse Leave**

Note

An update to the **Create** event of `obj_grid_block` does not need to be made to set `image_speed` to `0` as this is already taken care of by `scr_grid_setup`, which was created in Chapter 2, *Random Organization – Creating a Puzzle Game*.

Entering the grid with the mouse

When the **Mouse Enter** event is triggered by the player entering a grid block, the following script will be executed:

```
// Sets the column and row of the grid piece currently
being highlighted by the mouse.
global.grid_manager.current_mouse_col = col;
global.grid_manager.current_mouse_row = row;

// Sets the keyboard column and row information.
with (global.grid_manager)
{
    if (current_key_col >= 0 && current_key_row >= 0)
    {
        array_grid[current_key_col,
current_key_row].image_index = 0;
    }
    current_key_col = -1;
    current_key_row = -1;
}

// Sets the frame to the block's highlight state.
image_index = 1;
```

First, `current_mouse_col` and `current_mouse_row` are set within the global reference to the grid manager. These are not included in the following `with` statement because they reference the instanced variables `col` and `row` of `obj_grid_block`. Then, within the `with` statement that references the grid manager, the keyboard column and row are checked. If they are greater than or equal to zero—meaning that a grid block is currently being highlighted by the keyboard—that grid block's `image_index` value is set as the value of the idle frame and the indexes for the keyboard are set to `-1`. Finally, the `image_index` value of the grid block that triggered this event will be set to the idle frame.

Pressing the left mouse button

When the player presses the left mouse, the image index will be set to its swap frame, as shown in the following code:

```
// Sets the image_index to the down state.
image_index = 2;
```

Releasing the left mouse button globally

The **Global Left Released** event will be triggered by all instances of `obj_grid_block` when the left mouse button is released, as shown in the following code:

```
// If the mouse is currently over the current piece,
the piece is highlighted.
if (position_meeting(mouse_x, mouse_y, self))
{
    // Sets the image index to the image being
highlighted.
    image_index = 1;
}
else if (image_index == 1)
{
    // Sets the object back to the first frame, but
only if it waspreviously highlighted.
    image_index = 0;
}
```

First, the previous code checks whether or not the cursor is currently hovering over this grid block using the built-in function `position_meeting`, which tests to see whether or not a specified coordinate is contained within the requested instance's collision mask. If it is, the frame is set to the highlighted state. If the the user's mouse is not contained within a highlighted instance of `grid_block`, the `image_index` object is set to the first frame. The `image_index` object is checked to avoid setting the instances of `obj_grid_block` redundantly to the same `image_index` object.

The Mouse Leave event

The **Mouse Leave** event is probably the most important event of all four. This event will be used to swap pieces on the board. However, this will only work when the piece that the mouse leaves is in its swap state. A separate instanced Boolean could be created, but since `image_index` represents the current state of the grid, checking it is sufficient. The following is the code for this event's script:

```
/* If the grid block is currently being highlighted,
the pieces are swapped based on the direction the
mouse is moved.*/
if (image_index == 2)
{
    // Determine which way the blocks were swapped.
    var x_direction, y_direction;
    x_direction = mouse_x - x;
    y_direction = mouse_y - y;

    // Swap the pieces.
    scr_swap_pieces(col, row, x_direction,
y_direction, false);
}

// The image_index is set back to 0.
image_index = 0;
```

In the first part of this event's script, the object's `image_index` value is checked. If it is equal to `2`, meaning it is in the swap state, the pieces are swapped. First, the direction of the swap is derived and two local variables, `x_direction` and `y_direction`, are defined. By fetching the mouse's current coordinates and subtracting the position of the recently left object, the direction in which the mouse has moved can be determined. If a direction is negative, it has moved left or up with respect to `x` and `y`, and if it's positive, the mouse has moved to right or down. Then, these values as well as the column and row of the grid block that the mouse has recently left are calculated and used by `scr_swap_pieces`, which will be explained more thoroughly later in this chapter. Finally, the frame of the piece is set back to its idle `image_index` object as the piece is no longer highlighted or in the swap state.

Swapping pieces

Now that the mouse input has been integrated into the game, functionality to actually swap pieces can be implemented. The ability of the board to check for matches must also be changed a bit, and a new script to actually reorganize the board must be created as well.

obj_puzzle_piece

A minor update needs to be made to the **Create** event of `obj_puzzle_piece` involving the defining of the instanced variable `matched`, as shown in the following code:

```
/// Initializes variables for instances of
obj_puzzle_piece.

// Indicates whether the piece is currently adjacent
to other pieces.
matched = false;
```

The `matched` variable is used to indicate that a piece is currently adjacent to another piece. When removing matching pieces, the Boolean value that is obtained by `matched` will be checked.

scr_swap_pieces

This is a new script resource called by the **Mouse Leave** and **Mouse Enter** events of `obj_grid_block`. This script will swap two pieces on the board and then check the board for any matches.

The first portion of this script will initialize local variables and then determine the direction in which the pieces are to be swapped:

```
/* argument0 -- the column of the starting piece to be
swapped.
  argument1 -- the row of the start piece to be
swapped.
  argument2 -- the direction and value of the
horizontal movement.
  argument3 -- the direction and value of the vertical
movement.
  argument4 -- if true, the board will not be checked
again after the swapped; used to remedy improper
swaps.
 */

// Initializes two local variables, that will be
```

```
adjusted to the column and row to be swapped.
var other_col, other_row;
other_col = argument0;
other_row = argument1;

// If the horizontal difference is greater than the
vertical...
if (abs(argument2) > abs(argument3))
{
    // If argument2 is negative, the column is reduced
by one;
    // otherwise, it is increased.
    if (argument2 < 0)
    {
        other_col--;
    }
    else
    {
        other_col++;
    }
}
else
{
    // If argument3 is negative, the row is reduced by
one;
    // otherwise, it is increased.
    if (argument3 < 0)
    {
        other_row--;
    }
    else
    {
        other_row++;
    }
}
```

There are five arguments in this function. The first two represent the column and row of the piece to be swapped. The third and fourth arguments represent the horizontal and vertical directions of the movement, and the fifth argument is used to determine whether this is an incorrect swap. If the fifth argument is `true`, a check for matches will not be performed. There are then two local variables defined: `other_col` and `other_row`. These represent the indices of the column and row that the original variables will be swapped with.

Then, the horizontal and vertical directions derived from the **Mouse Leave** event are used to determine whether or not a column or row should be swapped and the direction in which it should be moved. This is done by first comparing the **absolute value** (`abs`) of `argument2` and `argument3`. The absolute value refers to the magnitude of a number regardless of its *sign*. More simply, when the `abs` function is used, it returns a positive number (excluding `0`, which is neither positive nor negative). If the absolute value of `argument2` is greater than that of `argument3`, a horizontal swap will occur; otherwise, a vertical swap will occur. The sign of the argument with the greater magnitude is then checked. If that value is less than `0`, the values of `other_col` or `other_row` will be decreased; otherwise, their respective indices will be increased.

After adjusting the values of `other_col` and `other_row`, the following code is executed:

```
// With the global instance of obj_grid_manager.
with (global.grid_manager)
{
    // Creates four local variables.
    var piece_a, piece_b, old_x, old_y;

    // Gets the current piece and other piece.
    piece_a = scr_get_puzzle_piece(argument0,
argument1);
    piece_b = scr_get_puzzle_piece(other_col,
other_row);

    // If either is null, the with statement is broken
out of.
    if (piece_a == noone || piece_b == noone)
    {
        break;
    }

    // stores the original piece's x and y values.
    old_x = piece_a.x;
    old_y = piece_a.y;

    // Sets the original piece's new column and row.
    piece_a.col = other_col;
    piece_a.row = other_row;
```

```
    // Sets the new piece's column and row.
    piece_b.col = argument0;
    piece_b.row = argument1;

    // Sets the original piece's x and y to the new
piece's.
    piece_a.x = piece_b.x;
    piece_a.y = piece_b.y;

    // Sets the new piece's x and y to the original
piece's.
    piece_b.x = old_x;
    piece_b.y = old_y;


    // Sets the piece reference within grid manager's
board.
    array_pieces[argument0, argument1] = piece_b;
    array_pieces[other_col, other_row] = piece_a;

    // If the fifth argument is true, the board is not
checked. Used for incorrect swaps.
    if (argument4)
    {
        break;
    }

    // Checks for matches.  If none were found, the
pieces are reset.
    if (!scr_check_board(true))
    {
        // The function is run again.
        scr_swap_pieces(argument0, argument1,
argument2,
          argument3, true);
    }
}
```

The previous block of code begins with a `with` statement that references the global instance of `obj_grid_manager`. Then, the following four new local variables are created:

- `piece_a`: This is the variable representing the original piece
- `piece_b`: This is the variable that represents the piece position received from the new position

- `old_x`: This is the variable that represents the original piece's `x` position
- `old_y`: This is the variable that represents the original piece's `y` position

The `piece_a` and `piece_b` variables are then assigned using `scr_get_puzzle_piece`, a script resource created in the previous chapter. `piece_a` represents the original piece and uses `argument0` and `argument1`; `piece_b` is the one to be swapped with `piece_a`, which uses `other_col` and `other_row`. If either of these pieces are invalid, for example, trying to swap the pieces to the left in the first column, this `with` statement is exited.

The positioning and the column and row values for the two pieces are swapped. The `old_x` and `old_y` variables are used to store the position of `piece_a` because if `piece_a.x` or `piece_a.y` were just immediately set, we would be unable to retrieve the original values. After information between the two pieces has been swapped, the references to each piece in the grid manager's two-dimensional array are swapped as well.

Then, if `argument4` is `true`, the `with` statement is exited; otherwise, `scr_check_board` is executed. If this execution returns `false`, the `scr_swap_piece` script is executed again with all the same arguments, except `true` is supplied for the fifth argument, making sure that the board is not checked another time and that this function isn't called again. In other words, if no match is found, the pieces are swapped again, returning them to their original positions.

The following is the entirety of this script:

```
/* argument0 -- the column of the starting piece to be
swapped.
  argument1 -- the row of the start piece to be
swapped.
  argument2 -- the direction and value of the
horizontal movement.
  argument3 -- the direction and value of the vertical
movement.
  argument4 -- if true, the board will not be checked
again after the swapped; used to remedy improper
```

```
swaps.
 */

// Initializes two local variables that will be
adjusted to the column and row to be swapped.
var other_col, other_row;
other_col = argument0;
other_row = argument1;

// If the horizontal difference is greater than the
vertical...
if (abs(argument2) > abs(argument3))
{
    // If argument2 is negative, the column is reduced
by one;
    // otherwise, it is increased.
    if (argument2 < 0)
    {
        other_col--;
    }
    else
    {
        other_col++;
    }
}
else
{
    // If argument3 is negative, the row is reduced by
one;
    // otherwise, it is increased.
    if (argument3 < 0)
    {
        other_row--;
    }
    else
    {
        other_row++;
    }
}

// With the global instance of obj_grid_manager.
with (global.grid_manager)
{
    // Creates four local variables.
    var piece_a, piece_b, old_x, old_y;

    // Gets the current piece and other piece.
```

```
    piece_a = scr_get_puzzle_piece(argument0,
argument1);
    piece_b = scr_get_puzzle_piece(other_col,
other_row);

    // If either is null, the with statement is broken
out of.
    if (piece_a == noone || piece_b == noone)
    {
        break;
    }

    // stores the original piece's x and y values.
    old_x = piece_a.x;
    old_y = piece_a.y;

    // Sets the original piece's new column and row.
    piece_a.col = other_col;
    piece_a.row = other_row;

    // Sets the new piece's column and row.
    piece_b.col = argument0;
    piece_b.row = argument1;

    // Sets the original piece's x and y to the new
piece's.
    piece_a.x = piece_b.x;
    piece_a.y = piece_b.y;

    // Sets the new piece's x and y to the original
piece's.
    piece_b.x = old_x;
    piece_b.y = old_y;

    // Sets the piece reference within grid manager's
board.
    array_pieces[argument0, argument1] = piece_b;
    array_pieces[other_col, other_row] = piece_a;

    // If the fifth argument is true, the board is not
checked.  Used for incorrect swaps.
    if (argument4)
    {
        break;
    }

    // Checks for matches.  If none were found, the
```

```
pieces are reset.
    if (!scr_check_board(true))
    {
        // The function is run again.
        scr_swap_pieces(argument0, argument1,
argument2,
          argument3, true);
    }
}
```

Updating scr_check_board

The `scr_check_board` script is the script resource created in the previous chapter that was used to randomize pieces using recursion upon initializing the game board. It is also executed in the recently written script, `scr_swap_pieces`, and is expected to return a Boolean; however, this functionality has not been implemented yet. This script will also be updated slightly to use the new variable matched from `obj_puzzle_piece`. If matches are found, it will then call a new script, `scr_reorganize_board`. Finally, it will return the Boolean expected for `scr_swap_pieces`.

The first addition to this script will involve the use of `argument0`. It is first used within the `while` statements that decrease the values of `h_count` and `v_count`, as shown in the following code:

```
if (!argument0)
{
  array_pieces[i + h_count, j].image_index =
irandom(piece_range);
}
else
{
  array_pieces[i + h_count, j].matched = true;
}
```

In the previous code block, if `argument0` is `false`, the `image_index` object of the puzzle piece is changed like it was in the previous chapter; otherwise, the new instanced variable of `obj_puzzle_piece`—`matched`—is set to `true`.

Then if `argument0` is `true`, this outermost `while` statement is exited with a `break` statement. Unlike when creating the board, the `while` statement

should only be executed once, regardless of whether or not matches were found. After exiting the statement, if `argument0` is `true` and a match was found, `scr_reorganized_board` is called; this will be explained soon. Finally, `match_found` is returned; this Boolean indicates whether or not a match has occurred and will be used in the function `scr_swap_pieces` to determine its next course of action. The following code performs all of this:

```
// The loop is exited if checking for matches to
break.
  if (argument0)
   {
       break;
   }
}

// If designated to destroy matching pieces, the board
is then reorganized.
if (argument0 && match_found)
{
    scr_reorganize_board();
}

// Returns whether or not any matches were found.
return match_found;
```

The following is the entirety of this script with the new additions highlighted:

```
/// Checks each piece on the board for matches.

/* argument0 -- if true, matching sets of three or
more are designated to be destroyed; otherwise,
matching sets are continuously adjusted.
*/

// Create a variable designating that matches have
been found.
var i, j, match_found;

// Set to true so the initial while loop runs at least
once.
match_found = true;
```

```
while (match_found)
{
    // match_found set to false.  If no matches are
found, the loop will be broken.
    match_found = false;

    // Loop through each column and row.
    for (i = 0; i < columns; i++)
    {
        for (j = 0; j < rows; j++)
        {
            // The count of adjacent, matching pieces
to the right and below.
            var h_count, v_count;
            h_count = scr_check_adjacent(i, j, 1, 0);
            v_count = scr_check_adjacent(i, j, 0, 1);


            // If the number of match pieces matching
to the right is greater than or equal to 2...
            if (h_count >= 2)
            {
                while (h_count >= 0)
                {
                    if (!argument0)
                    {
                        array_pieces[i + h_count,
j].image_index = irandom(piece_range);
                    }
                    else
                    {
                        array_pieces[i + h_count,
j].matched = true;
                    }
                    h_count--;
                }
                match_found = true;
            }

            // If the number of match pieces matching
below is greater than or equal to 2.
            if (v_count >= 2)
            {
                while (v_count >= 0)
                {
                    if (!argument0)
```

```
                    {
                        array_pieces[i, j +
v_count].image_index = irandom(piece_range);
                    }
                    else
                    {
                        array_pieces[i, j +
v_count].matched = true;
                    }
                    v_count--;
                }
                match_found = true;
            }
        }
    }

    // The loop is exited if checking for matches to
break.
    if (argument0)
    {
        break;
    }
}

// If designated to destroy matching pieces, the board
is then reorganized.
if (argument0 && match_found)
{
    scr_reorganize_board();
}

// Returns whether or not any matches were found.
return match_found;
```

Note

This code should not be tested yet as the `scr_reoganize_board` script has yet to be written.

Updating organization with scr_reorganize_board

The following script, called by `scr_check_board`, is used to destroy pieces that are part of matching sets. New pieces are then placed above those pieces that were shifted down. The checking process is then restarted in case the process of pieces being set in place has created new matching sets, as shown in the following code:

```
// Initializes various local variables.
var i, j, array_broken_piece, array_length, piece;

// Tracks the length of the array storing IDs of the
broken pieces.
array_length = 0;

// Loop through every piece on the board, looking for
pieces that have been designated as matching.
for (i = 0; i < columns; i++)
{
  for (j = 0; j < rows; j++)
  {
    if (array_pieces[i, j].matched)
    {
      // The local array is populated with
        the pieces that are matching.
      array_broken_piece[array_length++] =
        array_pieces[i, j];
    }
  }
}

// Loops through the newly created array.
for (i = 0; i < array_length; i++)
{
  // Assign the local variable since it will be used
many times.
  piece = array_broken_piece[i];

  // Sets matched to false so it can be checked again.
  piece.matched = false;
```

```
  // The piece is continually swapped with pieces
above it until
    it reaches the top.
  while (piece.row != 0)
  {
    // The fourth argument is -1 as it is known which
      direction the swap will take place.
    scr_swap_pieces(piece.col, piece.row, 0, -1,
true);
  }
  // Sets the image_index to give the illusion of a
new piece
    being created.
  piece.image_index = irandom(piece_range);
}

// Checks the board for new matches.
scr_check_board(true);
```

After initializing local variables, all of the pieces are looped through; if `matched` is `true` for a piece, it is added to the array, `array_broken_piece`. The recorded length of that array, `array_length`, is also incremented. Then, another `for` statement is executed in which each of the matched pieces are checked and moved up their column using `scr_swap_piece`. The `image_index` value of these pieces is then randomly assigned. Once all the previously matched pieces are shifted and randomized, the board is checked again for new matches.

Note

This script is called exclusively by `obj_grid_manager`, so its instanced variables can be referenced safely.

Integrating keyboard input

With the creation of the new scripts and events, the player should be able to highlight a grid block and swap pieces that create sets of three. What if the player wants to do this with the keyboard though? Normally, such requests go ignored, but allowing a player to use multiple input devices is a great way to expand a game's player base. The following sections will go over some new events and updates to the current scripts and events to integrate keyboard input with the game.

Introducing keyboard functions, variables, and events

Keyboard input can be accessed through a variety of ways. Keyboard-specific events are divided into the following three, separate types:

- **Keyboard**: This is an event triggered if the specified key is down
- **Key Press**: This is an event triggered if the specified key has just been pressed
- **Key Release**: This is an event triggered if the specified key has just been released

When adding one of these three events, every key that the player can press is accessible. The only unique ones are **<any key>**, which will trigger the respective **Keyboard** event regardless of the key that was pressed, and **<no key>**, which will trigger the respective **Keyboard** event if no key is active.

Keyboard inputs can also be detected through various functions described within the following code sample:

```
// Initialized variables.
var result, key;
result = "";

// The function, ord, converts a string value to its
```

```
ASCII character code.
key = ord('A');

// Checks whether the specified key is down on the
virtual or real keyboard.
if (keyboard_check(key))
{
    result = "A is down.";
}

// Checks whether the specified key was pressed.
if (keyboard_check_pressed(vk_enter))
{
    result = "Enter has been pressed.";
}

// Checks whether the specified key was released.
if (keyboard_check_released(ord('B')))
{
    result = "B was released.";
}
```

GameMaker: Studio also works with a virtual keyboard system, meaning that keys can actually be simulated. This can be useful for making the same key reproduce the same action without having to rewrite the script. This simulation is demonstrated in the following code sample:

```
// Simulates a press of the space bar.
keyboard_key_press(vk_space);

// Simulates a release of the space bar.
keyboard_key_release(vk_space);

// Checks whether the specified key was pressed, but
is not triggered through virtual keyboard functions.
if (keyboard_check_direct(vk_space))
{
    return true;
}
```

You may have observed by now that many keyboard functions take in arguments prefaced with `vk_`. These constants represent ASCII values of non-alphanumeric characters, such as the space bar or shift keys, that

cannot be created easily through the use of `ord`. A list of these constants can be found in GameMaker: Studio by navigating to **Help** | **Contents...**.

Integrating the Keyboard event updates

In this game, the player will use the arrow keys to navigate through the board. Then, holding either the *Shift* key or the space bar, they will put the currently highlighted grid block in its swap state. To integrate this keyboard functionality, several events must be added to `obj_grid_manager`.

Utilizing the press <any key> event

The code to be discussed in this section will be placed within the **Key Press** event of `obj_grid_manager` and triggered by any key.

The first portion of this code will handle the pieces that are highlighted by the mouse. If a grid block is highlighted by the mouse, the highlighted keyboard key values, `current_key_col` and `current_key_row`, are assigned the values of `current_mouse_col` and `current_mouse_row`. Then, `current_mouse_col` and `current_mouse_row` are set to negative values since the mouse is no longer the primary input being used

```
// If the mouse is highlighting a piece, this
highlight is carried over to the keyboard.
if (current_mouse_col >= 0 && current_mouse_row >= 0)
{
    current_key_col = current_mouse_col;
    current_key_row = current_mouse_row;

    current_mouse_col = -1;
    current_mouse_row = -1;
}
```

Then, if the keyboard column and row are in range (or greater than or equal to `0`), the currently highlighted piece is temporarily set back to the idle state. Following this, a local variable, `shift_down`, is initialized. This variable is a Boolean whose value is set to `true` if the space bar or one of the *Shift* keys are pressed. The following code makes the highlighted

piece go back to its initial frame:

```
// If a piece is currently highlighted by the
keyboard, this piece goes back to its initial frame.
if (current_key_col >= 0 && current_key_row >= 0)
{
    array_grid[current_key_col,
current_key_row].image_index = 0;
}
else
{
  // The currently highlighted piece is out of range,
the top
    left is assigned.
  current_key_col = 0;
  current_key_row = 0;
}

// Variable assigned if either shift key or the space
bar is down.
var shift_down = keyboard_check(vk_shift) ||
  keyboard_check(vk_space);
```

A `switch` statement is then introduced, which uses `keyboard_key`. This read-only variable represents the ASCII value of the recently pressed key. There are only four cases used in this `switch` statement, as shown in the following list:

- `vk_up`
- `vk_down`
- `vk_left`
- `vk_right`

These cases represent the respective arrow keys. Within each case, `current_key_col` and `current_key_row` are incremented by the appropriate amount; however, if `shift_down` is `true`, the pieces are swapped as well. The following is the code for these cases:

```
// Switch statement used for the four main keyboard
directions.
switch(keyboard_key)
{
```

```
  case vk_left:
    if (shift_down)
    {
      scr_swap_pieces(current_key_col,
current_key_row,
        -1, 0, false);
    }
    current_key_col--;
    break;
  case vk_right:
    if (shift_down)
    {
      scr_swap_pieces(current_key_col,
current_key_row,
        1, 0, false);
    }
    current_key_col++;
    break;
  case vk_up:
    if (shift_down)
    {
      scr_swap_pieces(current_key_col,
current_key_row,
        0, -1, false);
    }
    current_key_row--;
    break;
  case vk_down:
    if (shift_down)
    {
      scr_swap_pieces(current_key_col,
current_key_row,
        0, 1, false);
    }
    current_key_row++;
    break;
  }
```

Each case could be its own **Key Press** event, but placing one in the `switch` statement makes it more customizable. If you want to use other keys, this `switch` statement can easily be edited.

Then, `current_key_col` and `current_key_row` are kept within the range of the board using the built-in function `clamp`. This function makes sure that a given value does not fall outside the bounds of a specified range, which

is in this case, the board's columns and rows. Finally, the updated grid block is highlighted to the appropriate `image_index` object: `2` if `shift_down` is `true` and `1` if it is `false`.

The following is the entirety of this event's script:

```
// The current key clamped to make sure an out of
range grid block is not highlighted.
current_key_col = clamp(current_key_col, 0, columns -
1);
current_key_row = clamp(current_key_row, 0, rows - 1);

// If shift or space are down, the current grid block
goes to the down frame; otherwise, just the highlight
frame.
if (shift_down)
{
    array_grid[current_key_col,
current_key_row].image_index = 2;
}
else
{
    array_grid[current_key_col,
current_key_row].image_index = 1;
}

// If the mouse is highlighting a piece, this
highlight is carried over to the keyboard.
if (current_mouse_col >= 0 && current_mouse_row >= 0)
{
    current_key_col = current_mouse_col;
    current_key_row = current_mouse_row;

    current_mouse_col = -1;
    current_mouse_row = -1;
}
// If a piece if currently highlighted by the
keyboard, this piece goes back to its initial frame.
if (current_key_col >= 0 && current_key_row >= 0)
{
    array_grid[current_key_col,
current_key_row].image_index = 0;
}
else
{
```

```
  // The currently highlighted piece is out of range,
the top
    left is assigned.
  current_key_col = 0;
  current_key_row = 0;
}

// Variable assigned if either shift key or the space
bar is down.
var shift_down = keyboard_check(vk_shift) ||
  keyboard_check(vk_space);

// Switch statement used for the four main keyboard
directions.
switch(keyboard_key)
{
  case vk_left:
    if (shift_down)
    {
      scr_swap_pieces(current_key_col,
current_key_row,
        -1, 0, false);
    }
    current_key_col--;
    break;
  case vk_right:
    if (shift_down)
    {
      scr_swap_pieces(current_key_col,
current_key_row,
        1, 0, false);
    }
    current_key_col++;
    break;
  case vk_up:
    if (shift_down)
    {
      scr_swap_pieces(current_key_col,
current_key_row,
        0, -1, false);
    }
    current_key_row--;
    break;
  case vk_down:
    if (shift_down)
    {
      scr_swap_pieces(current_key_col,
```

```
current_key_row,
        0, 1, false);
    }
    current_key_row++;
    break;
  }

// The current key clamped to make sure an out of
range grid block is not highlighted.
current_key_col = clamp(current_key_col, 0, columns -
1);
current_key_row = clamp(current_key_row, 0, rows - 1);

// If shift or space are down, the current grid block
goes to the down frame; otherwise, just the highlight
frame.
if (shift_down)
{
    array_grid[current_key_col,
current_key_row].image_index = 2;
}
else
{
    array_grid[current_key_col,
current_key_row].image_index = 1;
}
```

Implementing the release <any key> event

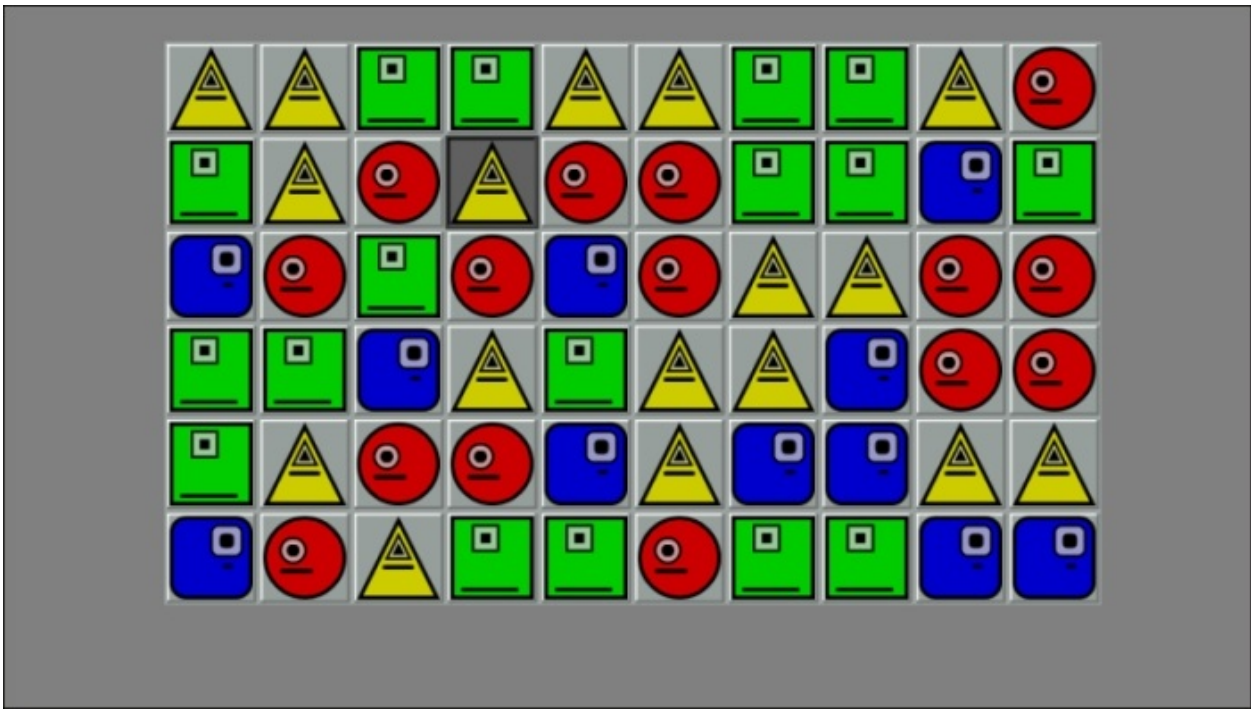
Right now, if the game were to be started, the player would notice that the block highlighted with the keyboard remains in the swap state upon releasing the *Shift* key or the space bar. To remedy this, a **Key Release** event for any key will be added to `obj_grid_manager` with the following script:

```
// If the current key column and row are in range...
if (current_key_col >= 0 && current_key_row >= 0)
{
    // If space bar and neither shift key are not
down, the current grid block goes back to its
highlight frame.
    if (!keyboard_check(vk_space) &&
!keyboard_check(vk_shift))
    {
```

```
        array_grid[current_key_col,
    current_key_row].image_index
        = 1;
        }
    }
```

This event simply changes the state of the currently highlighted grid block, but only if the space bar and *Shift* are not held down. Again, this event could be separated into two, but having one makes the game easier to customize and less rigid.

That's it! If integrated properly, the player should now be able to navigate the board using only the keyboard, and swap pieces by holding down the space bar or the *Shift* key. The following is a preview of what the board should look like with one grid block on the board in its swap state:



Summary

In this chapter, two of the most common types of input in GameMaker: Studio were discussed: mouse and keyboard input. Differences between the types of **Mouse** events were explained and then integrated into the puzzle game introduced in the previous chapter.

To complete the keyboard integration, however, two new scripts were created: `scr_swap_pieces` and `scr_reorganize_board`, which are used to swap pieces and then reorganize the board if a matching set of three or more pieces is made.

Then, information about keyboard events and functions was discussed. Using the **Key Press** and **Key Release** events, keyboard functionality was integrated into `obj_grid_manager`, so the player could navigate the board with the keyboard as well as the mouse.

There is one issue with this current setup though; when the player swaps pieces, all swapping and reorganization happens instantaneously! The player is unable to see exactly what has happened, or why something has happened. This is especially problematic when one encounters chains of breaks. What the player really needs is feedback! In the next chapter, several topics such as alarms, particle effects, and sound effects will be introduced in the hope of making this feedback more clear and making the game more understandable and enjoyable.

Chapter 4. Juicy Feedback – Aural and Visual Effects

In Chapter 3, *So How Do I Play? – Adding Player Interaction*, the player was given the ability to swap puzzle pieces on the board in the hope of creating matches; however, a problem was introduced. The player received no clear way of knowing what actions they performed or why something did (or didn't) work.

What players are missing is feedback, that is, indicators to let them know they are doing something successfully or unsuccessfully. This feedback can come in a variety of forms: sound effects, particle effects, and even just slight delays and animated movement. These can let the player know how they are doing. Also, the "juicier" the feedback is, the more polished the game will look.

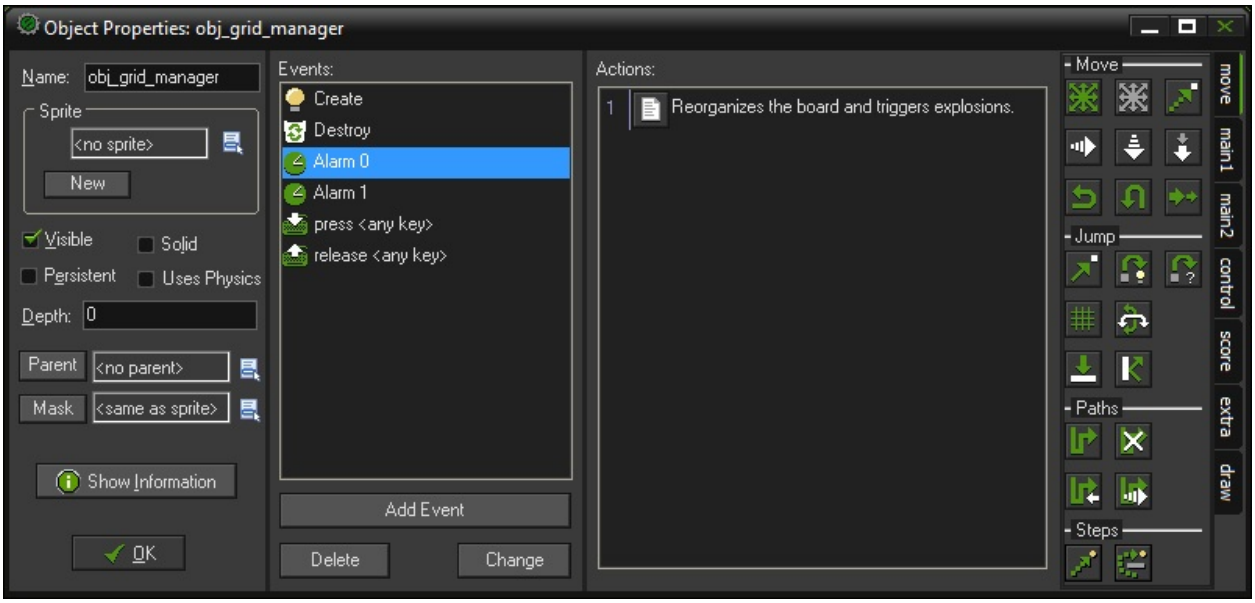
This chapter will focus on feedback in the following three ways:

- Using alarms to delay actions
- Audio effects
- Particle effects

Using these three approaches, feedback and visual flare will be added to the puzzle game with the goal of making it more clear and understandable to potential players.

Introducing alarms

Alarms in GameMaker: Studio are events that can be set up to occur after a specified number of frames have passed. Each object resource can have up to 12 alarms each, all of which are accessed through a built-in instanced array aptly named `alarm`. They can be added to objects within the **Object Properties** panel under **Add Event** | **Alarm**. The following is a screenshot of two **Alarm** events that are present in `obj_grid_manager`:



Arming the puzzle game's alarms

No new resources need to be constructed to integrate alarms with the puzzle game, but several updates must be made throughout the existing code. The goal of using alarms is to delay the events that occur when swapping pieces. Firstly, the pieces being swapped should translate from one position to another. Then, after a set of three matching pieces is made, they should disappear. The remaining and new pieces should then fall into new positions instead of jarringly pop like before. The updates and new **Alarm** events added to the the existing events and scripts will aid in doing this.

Setting up the first alarm

The first **Alarm** event created will be applied to `obj_grid_manager`. This alarm will execute only one script and have only one line of actual code. This should be assigned to **Alarm 0** as that is the alarm that will be referenced later, as shown in the following code:

```
/// Reorganizes the board and triggers explosions.

scr_reorganize_board();
```

Within this simple script, `scr_reorganize_board` is being executed. The `scr_reorganize_board` script was introduced in the previous chapter, and it deals primarily with the replacement and reassignment of pieces that are part of a matching set and moving pieces into new spots. `scr_reorganize_board` is being executed by the alarm so the reorganization of the board can be slightly delayed.

Are pieces shifting?

One issue with alarms is that while waiting for one to trigger, the player can interact with the game and may create undesirable bugs. Adding the following script to the **Create** event of `obj_grid_manager` will prevent the player from swapping pieces while pieces on the board are visually shifting:

```
/// Initializes variables for the state of an alarm-
based break.

// Indicates that pieces are being shifted.
is_shifting_pieces = false;
```

Again, the newly created Boolean, `is_shifting_pieces,` will be used to prevent the player from swapping pieces while they are shifting or falling into place. This will, in turn, prevent visual confusion.

Applying is_shifting_pieces

After this addition of `is_shifting_pieces` to the **Create** event of `obj_grid_manager`, this variable will be be utilized in the following two events:

- The **<Any key>** event from the **Key Press** event of `obj_grid_manager`
- The **Mouse leave** event of `obj_grid_block`

In the **<Any key>** event from the **Key Press** event of `obj_grid_manager`, the local variable `shift_down` is adjusted. These adjustments are highlighted in the following code:

```
// Variable assigned if either shift key or the
spacebar is down.
var shift_down;
shift_down = !is_shifting_pieces &&
(keyboard_check(vk_shift) ||
keyboard_check(vk_space));
```

Earlier, `shift_down` was set based on whether or not a *Shift* key or the space bar was being held down. Now, the two executions of `keyboard_check` are encased within a set of parentheses. This will result in one Boolean which is then compounded with `is_shifting_pieces`. The `shift_down` variable will now only be `true` if `is_shifting_pieces` is `false`, regardless of the player having held down the space bar or *Shift* keys.

The alterations in the **Mouse leave** event of `obj_grid_block` are similar. They are highlighted in the following code:

```
/* If the grid block is currently being highlighted
and pieces are not shifting, the pieces are swapped
based on the direction the mouse has moved. */
if (image_index == 2 &&
!global.grid_manager.is_shifting_pieces)
{
    // Determine which way the blocks were swapped.
    var x_direction, y_direction;
    x_direction = mouse_x - x;
    y_direction = mouse_y - y;

    // Swap the pieces.
    scr_swap_pieces(col, row, x_direction,
y_direction, false, 0);
}
```

Earlier, if the `image_index` value of the `obj_grid_block` was `2`, meaning it was in the swap state, pieces would swap upon the mouse pointer leaving the location. However, now, even if the pieces are in the swap state, they will not swap if pieces are already shifting on the board.

Determining where pieces should move

Previously, `is_shifting_pieces` was created to prevent pieces from being swapped while others are being moved. Now, two new variables will be introduced into the **Create** event of `obj_puzzle_pieces`, directing instances as to where to go upon being swapped, as shown in the following code:

```
/// Initialization for variables involved with puzzle
piece movement.

// Goal position of the puzzle piece on the x.
x_goal = x;

// Goal position of the puzzle piece on the y.
y_goal = y;
```

The two variables, `x_goal` and `y_goal`, represent the corresponding coordinates that the puzzle pieces will move towards. They are set to the object's current `x` and `y` positions so the object will not need to move on instantiation.

Setting the Alarm 0 event of obj_puzzle_piece

Now that the two important variables, `x_goal` and `y_goal`, have been established, two **Alarm** events will be added to `obj_puzzle_piece`. The **Alarm 0** event of `obj_puzzle_piece` will smoothly move a piece to its goal position using the following code:

```
/// Moves puzzle piece towards goal smoothly.

// Linear interpolation of the x coordinate.
x = lerp(x, x_goal, 0.2);

// Linear interpolation of the y coordinate.
y = lerp(y, y_goal, 0.2);

// If the x or y are more than 1 unit away from the
goal, the alarm
// is set to run again next frame.
if (abs(x - x_goal) >= 1 || abs(y - y_goal) >= 1)
{
    alarm[0] = 1;
```

```
}
else
{
    x = x_goal;
    y = y_goal;
}
```

The first part of the code deals with using the built-in function `lerp`, which stands for linear interpolation, and is used to move the current position towards the specified goal value. Linear interpolation is a method of getting values in relation to two points using a percentage. Using `0.2` as the third value means that every time the **Alarm** event is triggered, the piece will move towards the goal position by about 20 percent of the distance between its current and goal positions. Using `0` as the third value would make the piece not move at all; meanwhile, using `1` would make the piece instantly snap to the goal position.

After moving the puzzle piece towards its goals, the distance of the goal from each coordinate is checked by comparing the absolute value of the goal (`abs`) and the current position to `1`. If either the `x` or `y` value is more than `1` unit away, the **Alarm** event, which in this case, is indexed at `0`, is reset to `1`; this means that one frame later, the alarm will be triggered once again. However, if the piece is within the range of its goal position, its coordinates are set to those of the goal position.

Note

There is a built-in function known as `distance_to_point`, but this function checks the distance of the object's bounds to the specified piece, which in this situation, would result in inaccurate placement of the piece.

Now a **Step** event could be used to perform this movement; however, this means that every piece would try to move to its goal position every frame, and since the goal position and main positions will match for the most part, there is no need to constantly update the pieces' positions; so, utilizing an alarm can perform this much more efficiently.

Making pieces fall with Alarm 1 in obj_puzzle_piece

Similar to **Alarm 0**, **Alarm 1** will also focus on the movement of the pieces; however, **Alarm 1** will focus on making a piece fall into a new spot, such as when a new piece is generated above the game board or when a piece from below has been removed. The following is the code for **Alarm 1**:

```
// Moves the puzzle piece towards the y goal using
gravity.
gravity = 1;

/* If the current position plus the speed is greater
than the goal,
  the piece is stopped; otherwise, the alarm is reset.
/
if (y + speed >= y_goal)
{
    y = y_goal;
    speed = 0;
    gravity = 0;
}
else
{
    alarm[1] = 1;
}
```

In this event, the built-in instanced variable `gravity` is first set to `1`, meaning the object's speed will increase by one unit per frame, making it accelerate as it falls. By default, setting `gravity` will cause an object to fall vertically. Then, if the sum of `y` and `speed` is greater than or equal to the value of `y_goal`, the `gravity` and `speed` values are set to `0`, and the object's `y` position is set to `y_goal`. The sum is used like a prediction. After this event is triggered, the object is moved by its `speed` value; if just `y` positions were checked and not the sum of `y` and `speed`, there would be one frame where the object would fall past the value of `y_goal` and then snap back to the proper position in the next frame. On the other hand, if the previously mentioned sum is less than `y_goal`, the value of `alarm[1]` will be set to `1`, making it rerun the next frame. This small pop can be

rather jarring, and preventing it makes the game appear more polished.

> **Note**
>
> By default, setting an object's `gravity` object to a positive value will cause it to fall vertically; however, if the built-in variable `gravity_direction` is changed, which defaults to `270`, this will not be true.

Setting the alarms

In the previous section, two alarm events for `obj_puzzle_piece` were defined; however, the values for these alarms are not assigned outside of `obj_puzzle_piece`. Some other object has to call them. In this section, previously written scripts, `scr_swap_pieces`, `scr_check_board`, and `scr_reorganize_board`, will be altered, calling the alarms and other variables created earlier.

Updating scr_swap_pieces

The updates in `scr_swap_pieces` involve a new argument as well as swapping `x_goal` and `y_goal` between pieces, as shown in the following code:

```
/* argument0 -- the column of the starting piece to be
swapped.
  argument1 -- the row of the start piece to be
swapped.
  argument2 -- the direction and value of the
horizontal movement
  argument3 -- the direction and value of the vertical
movement
  argument4 -- if true, the board will not be checked
again after the swapped; used to remedy improper
swaps.
 * argument5 -- which alarm to use for swapping
pieces, 0 for swaps, 1 for falling.
 */
```

The first update is made just to the comment, but it is still important to note that a new argument is being used. The next set of updates, as shown in the following code, are not utilized until the `with` statement is created several lines after the initial section:

```
...
// With the global instance of obj_grid_manager.
with (global.grid_manager)
{
    // Creates four local variables.
    var piece_a, piece_b, old_x_goal, old_y_goal;

    // Gets the current piece and other piece.
    piece_a = scr_get_puzzle_piece(argument0,
argument1);
    piece_b = scr_get_puzzle_piece(other_col,
other_row);

    // If either is null, the with statement is broken
out of.
    if (piece_a == noone || piece_b == noone)
    {
        break;
    }

    // stores the original piece's x and y values.
    old_x = piece_a.x;
    old_y = piece_a.y;

    // Stores the original piece's x_goal and y_goal.
    old_x_goal = piece_a.x_goal;
    old_y_goal = piece_a.y_goal;

    // Stores the original piece's column and row.
    piece_a.col = other_col;
    piece_a.row = other_row;

    // Sets the new piece's column and row.
    piece_b.col = argument0;
    piece_b.row = argument1;

    // Sets the new piece's x and y goals to the
original piece's x and y goals.
    piece_a.x_goal = piece_b.x_goal;
```

```
    piece_a.y_goal = piece_b.y_goal;
piece_b.x_goal = old_x_goal;
    piece_b.y_goal = old_y_goal;

    // Sets the piece reference within grid manager's
board.
    array_pieces[argument0, argument1] = piece_b;
    array_pieces[other_col, other_row] = piece_a;

    // Sets the alarm based on the fifth argument.
    piece_a.alarm[argument5] = 1;
    piece_b.alarm[argument5] = 1;

    // If the fifth argument is true, the board is not
checked.  Used for incorrect swaps.
    if (argument4)
    {
        break;
    }

    // Checks for matches.  If none were found, the
pieces are reset.
    if (!scr_check_board(true))
    {
        // The function is run again.
        scr_swap_pieces(argument0, argument1,
argument2,
        argument3, true, 0);
    }
    else
    {
        // Designates that the board is shifting
pieces.
        is_shifting_pieces = true;
    }
}
```

The important updates have been highlighted. Previously, the `x` and `y` coordinates of pieces were swapped; however, now the `x_goal` and `y_goal` values of pieces are also swapped. Then, the alarm at the index of `argument5`, is set to `1` so that it starts running the next frame.

An additional argument has also been added to the execution of `scr_swap_pieces`. The value `0` is used in the previous code since this execution represents incorrect swaps. This must be done to the other two

executions of `scr_swap_pieces` in the **<Any key>** event of the **Key Press** event of `obj_grid_manager` and the **Mouse leave** event of `obj_grid_block`. Not doing so will result in compile errors.

Finally, if a match is found, designated by the Boolean returned from the `scr_check_board` script, `is_shifting_pieces` within the global reference of `obj_grid_manager` is set to `true`, which temporarily prevents interfering swaps.

Updating scr_check_board

The update to `scr_check_board` is fairly simple as shown in the following code:

```
// If designated to destroy matching pieces, the board
is then reorganized.

if (argument0 && match_found)
{
    // Sets an alarm to organize the board.
    alarm[0] = 45room_speed * 1.5;
}

// Returns whether or not any matches were found.
return match_found;
```

First, if `argument0` is `true`—meaning the game is checking for matches—and a match was found, `alarm[0]` is set to a value 1.5 times that of the `room_speed` value. The variable `room_speed` is a built-in variable that refers to the frame rate of the current room; multiplying the `room_speed` value by 1.5 should make the alarm run for about one and a half seconds regardless of the `room_speed` value. This should be enough time to complete the swap being performed before pieces are broken and moved around when `scr_reorganize_board` is executed by `obj_grid_manager`.

Updating scr_reorganize_board

Finally, the `scr_reorganize_board` function that is called to reorganize a piece is updated, primarily to set the alarm for broken pieces so that they

can fall into place. These updates are made to the `for` statement called when recursively going through all of the matched pieces and highlighted in the following block of code:

```
// Loops through the newly created array.
for (i = 0; i < array_length; i++)
{
    // Assign the local variable since it will be used
frequently.
    piece = array_broken_piece[i];

    // Sets matched to false.
    piece.matched = false;

    // Move the piece off-screen.
    piece.y = -100;

    // The piece is continually swapped with pieces
above it until it reaches the top.
    while (piece.row != 0)
    {
        // Continually move the piece up so the pieces
fall in order.
        piece.y -= 100;

        // The fourth argument is -1 as it is known
whichdirection the swap will take place.
        scr_swap_pieces(piece.col, piece.row, 0, -1,
true, 1);
    }

    // Set first alarm to 1 so pieces at row 0 will
still fall intoplace.
    piece.alarm[1] = 1;

    // Sets the image_index to give the illusion of a
new piecebeing created.
    piece.image_index = irandom(piece_range);
}

// Checks the board for new matches.
if (!scr_check_board(true))
{
    is_shifting_pieces = false;
}
```

There are several important changes to note in the previous code. The first is that all broken pieces are initially moved to `-100` on the *y* axis, which will move them off screen. Then, for every swap upward, the piece is moved using `scr_swap_pieces`; the value of `piece.y` is reduced by another 100 units. This is to give the appearance of the pieces falling in order when they come back onto the board.

Also, while moving the broken pieces up the board, the sixth argument of `scr_swap_pieces` is now provided with `1`, so each broken piece sets the timer value of `alarm[1]` instead of `alarm[0]`, making the pieces fall and accelerate instead of smoothly interpolating into place.

Then, the timer value of `alarm[1]` for each broken piece is set to `1`. This is to make sure that broken pieces located at the topmost row still fall into place since they have not had to swap with any pieces but have still been moved off the screen.

Finally, the board is checked again for matches, and only if no matches are found, the `is_shifting_pieces` function is set to `false`, giving control back to the player.

If implemented properly, the player should now be able to swap pieces on the board and watch them smoothly transition from one position to another. Then, after a short delay, any matching sets should disappear from the board and pieces should fall into place. The integration of these alarms is a great step towards making the game clearer and livelier, but more can be done! In the next section, audio resources will be created and implemented in the game to give feedback in a different way.

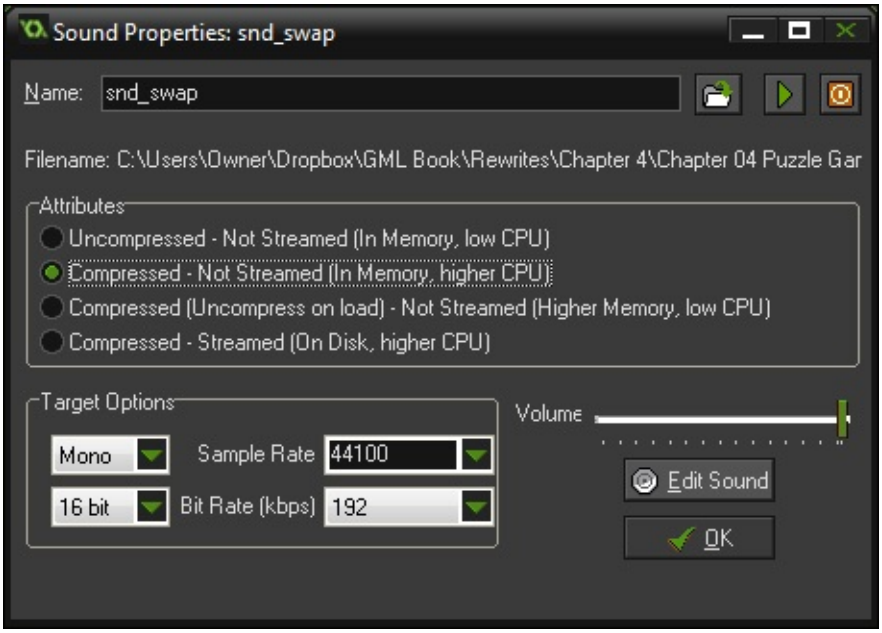
Hiding the pin drop – adding audio

Right now, the pieces in the puzzle game can be seen shifting as they swap with one another and fall into place. One type of feedback still missing is sound. The game is completely silent.

In the following sections, sound resources will be created and GML functions related to audio will be explained. Then, the newly created sounds will be integrated with the puzzle game.

Creating sound resources

Sound resources can be created by navigating to **Resources** | **Create Sound** or by pressing *Ctrl* + *Shift* + *U*. The **Sound Properties** dialog box is displayed in the following screenshot:



The button with the folder icon will bring up a file selection dialog in which an MP3 (`*.mp3`**)** or Wave (`*.wav`) file can be loaded. It is recommended that you use Wave files for sound effects and MP3 files for music. This is due to size and audio fidelity considerations. Wave files tend to be cleaner than MP3 files since they are usually uncompressed, but because of this, they tend to take up more space. This is fine for short clips of audio, but for songs that can last several minutes, MP3 is the more appropriate choice. Once the file is loaded, the sound clip can be played using the green arrow to the right of the previously mentioned button in which it will loop; the red stop button adjacent to the arrow will stop the sound clip from playing.

More information about the different settings displayed in this window can be found in GameMaker: Studio's help contents. For now, all sound effects should use the second option, **Compressed - Not Stream**, and all music should use the first option, **Uncompressed - Not Streamed**. All

Target options can be left as they are.

Gathering audio resources

In this game, four sounds will be created: three sound effects and one background music track, as shown in the following list:

- `snd_swap`: This sound will be used when pieces are being swapped, telling the players that they have successfully swapped two pieces. It should be a smooth, short shuffling sound.
- `snd_incorrect_swap`: This sound will be used when the player tries to swap pieces but does not make a match. This sound should be sharp and vitriolic, similar to the sound the buzzer game produces when a player gives an incorrect answer.
- `snd_pop`: This sound will be used when pieces are actually destroyed and moved off the screen. This should be something playful and light, yet not grating or annoying since the player will be hearing it a lot.
- `bgm_normal`: This is the music track. It can be anything really since the music isn't serving the purposes of feedback but instead eliminating complete silence while keeping the player engaged.

Introducing audio functions

Before integrating the four sound resources described previously, the built-in audio functions should be explained. As of this writing, recent versions of GameMaker: Studio use an updated audio engine. Functions for the old audio engine still persist, but these will not be covered in this book as there is a risk that they will eventually be deprecated and phased out of GML entirely. All of these new functions are prefaced with `audio_`. The following code sample describes a majority of the built-in audio functions that deal with sound, which are closely related to those that will be used in this chapter:

```
/* Sets the number of audio channels or simultaneous
sounds that can be played.  If the number of sounds
being played exceeds this, sounds are played based on
```

```
their priority.  The default number of channels is
128.  Using fewer channels can increase performance.
audio_channel_num(92);

 Gets the type of the sound.
  0:  Sound effect
  1:  Background music
 * -1:  Error .
var sound_type = audio_get_type(bgm_normal);

 Plays a sound.
  argument0 is the sound's resource ID.
  argument1 is the priority.  Higher priority sound
will play over lower ones if the number of available
sound channels becomes low.
  argument2 indicates if the sound loops or not.
  An ID is returned that can be edited with other
built-in functions.
 /
var new_sound = audio_play_sound(snd_pop, 0, false);

// Returns the length of the given sound in seconds.
audio_sound_length(new_sound);

/ Adjusts the pitch of a given sound.
  argument0 is the sound ID.
  argument1 is the pitch, which should range between 0
and 5, defaulting at 1.
  A higher value will make the playing sound more high
pitched.
  A lower value will give the sound a lower, deeper
pitch.
 * A sound resource ID can also be used, which will
change the pitch whenever said sound is played.

audio_sound_pitch(new_sound, 0.5);

 Fades in a sound effect.
  argument0 is the sound ID.
  argument1 is the volume, which should range between
0 and 1.
  argument2 is the amount of time the sound takes to
reach the goal volume in milliseconds.
  The example below will fade the newly created sound
to complete silence over 4 seconds. */
audio_sound_gain(new_sound, 0, 4000);
```

```
// Pauses the given sound.
audio_pause_sound(new_sound);

// Resume the given sound if paused.
audio_resume_sound(new_sound);

// Stops the given sound.
audio_stop_sound(new_sound);
```

There are more advanced sound functions to dynamically affect the volume and panning of sounds; these will be explained in future chapters.

This following code sample highlights some of the audio effects used for playing music; only one music track can be played at any given time:

```
// Plays the given music sound resource.  The second
argument specifies if the music should loop or not.
audio_play_music(bgm_normal, true);

// Returns a boolean specifying if music is playing.
audio_music_is_playing();

// Pauses the music that is playing.
audio_pause_music();

// Resumes the music if it has been paused.
audio_resume_music();

/* Fades in music.
  argument0 is the volume value that the music fades
in to.
  argument1 is the amount of time in milliseconds to
reach said volume.
  In the below, the music being played would take
about ten seconds to fade to silence. /
audio_music_gain(0, 10000);

// Stops the currently playing music.
audio_stop_music();

// Pauses all sounds and music playing.
audio_pause_all();

// Resumes all paused sound and music.
audio_resume_all();
```

```
// Stops all sounds and music.
audio_stop_all();

/* Sets the master volume of the game for all sounds
and music.
  argument0 is the percentage volume from 0 to 1. /
audio_master_gain(1);
```

> **Note**
>
> By default, the new audio engine is enabled through the **Global Game Settings** dialog window, which can be opened by going to **Resource** | **Change Global Game Settings** or by pressing *Shift* + *Ctrl* + *G*. There, a box labeled **Use New Audio Engine** is checked. Unchecking this will cause the functions previously described to get disabled.

Playing puzzling sounds

In this section, the audio functions will be added to the existing set of scripts and events that have been previously written.

Swapping sounds in scr_swap_pieces

At the end of the script resource `scr_swap_pieces`, the audio for swapping pieces will be integrated as shown in the following code:

```
// Checks for matches.  If none were found, the pieces
are reset.
    if (!scr_check_board(true))
    {
        // The function is run again.
        scr_swap_pieces(argument0, argument1,
argument2, argument3, true);

        // An incorrect swap sound effect is played.
        audio_play_sound(snd_incorrect_swap, 0,
false);
```

```
    }
    else
    {
        // Plays a swap sound to signify that a swap
has properly occurred.
        audio_play_sound(snd_swap, 0, false);

        // Designates that the board is shifting
pieces.
        is_shifting_pieces = true;
    }
```

In this section, if the board is being checked for matches but none are found, the incorrect sound `snd_incorrect_swap` is played; on the other hand, if the matches are found, `snd_swap` should be played. Neither of these sounds requires high priority nor should the sound loop, so `0` and `false` are supplied for the second and third arguments in both sounds.

Playing the pop sound in scr_reorganize_board

Only one line needs to be added to `scr_reorganize_board`, as shown in the following snippet:

```
// Plays the pop noise.  Only played once to avoid
noise.
audio_play_sound(snd_pop, 0, false);
```

The previous line of code can be placed immediately after defining `array_length`. The idea is that the pop sound should be played only once. If it were played for every piece that is broken, there would be a lot of noise, which would be very unpleasant.

Fading in the music with obj_grid_manager

In the **Create** event of `obj_grid_manager`, the following code will be added after `is_shifting_pieces` is initialized:

```
// Starts playing the music in silence.
audio_music_gain(0,0);
```

```
// Plays the music, starting in silence.
audio_play_music(bgm_normal, true);


// Fades the music to full volume in 10 seconds.
audio_music_gain(1, 10000);
```

Firstly, the music volume is set to `0`. This is so that the music can fade in. Then, the sound resource `bgm_normal` is played and looped. The music is then faded in using `audio_music_gain`.

Using this code, the game's main music should fade in to full volume in about 10 seconds. If all of these sounds and scripts have been integrated properly, the player should be able to hear all of the different sounds, giving the player more types of feedback and make the game more engaging.

Translating the pieces and sound effects is nice, but the game could still use some pizzazz—some juiciness! In the next section, particle effects will be introduced and placed in the game.

Visualizing effects with particles

So far the puzzle game has been given the functionality to add some delays in the gameplay so that pieces can be shown as moving through space. Audio has also been added, but the game could still use some sprucing up! This is where particle effects come in. Particle effects are a special class of objects in GameMaker: Studio. They are used to create a variety of effects, such as fire, explosions, sparks, rain, and snow. Some of these particle effects are environmental, immersing the player further into the game, but sometimes—as with this game—they are used to give feedback.

Particle effects are created using several components, as shown in the following list:

- **Particle systems**: This is an autonomous system that manages the updating and rendering of the current particles
- **Particle emitters**: This component manages the area in which particles are created or emitted from
- **Particle types**: These are the defining parameters of a particle effect, such as color, shape, lifetime, orientation, and movement

These components are created and managed through a variety of functions, which will be explained in the following sections.

Managing a particle system

The creation of a particle effect relies on the utilization of many different built-in functions. The following code sample showcases all of the different functions associated with the creation and management of particle systems:

```
// Creates a new particle system, returning its ID.
ps = part_system_create();

/* Sets the depth of the particle system.
  argument0 is the ID of the particle system.
```

```
  argument1 is the depth, similar to object's depth. /
part_system_depth(ps, 0);

// Clears the system of all particles and emitters
associated with the system.  Also resets all particle
systems properties such as position, if changed.
part_system_clear(ps);

// Returns true if a particle exists at the given ID;
otherwise, false is returned.
part_system_exists(ps);

/ Determine whether the given particle system will
draw new particles on top of old particles or the
other way around.
  argument0 is the particle system ID.
  argument1, if true, old particles are drawn on top
of new ones; false for the opposite.
part_system_draw_order(ps, true);

 Specifies whether the particle system should update
by itself
  argument1, if true, the particle system will update
the particles automatically; otherwise it won't. /
part_system_automatic_update(ps, false);

//Similar to part_system_automatic_update, except it
determines whether the particles will be rendered
automatically or not.
part_system_automatic_draw(ps, false);

// Updates the particle system.  Can be used to fast
forward a particle system, or to update the particle
system if it does not automatically update.
part_system_update(ps);

// Draws the particle system, mostly used to render a
particle system that is not being drawn automatically.
part_system_drawit(ps);

/* Moves the particle system.
  argument0 is the particle system ID.
  argument1 is the x-coordinate.
  argument2 is the y-coordinate.
  Moves all particles and emitters associated with the
system. */
part_system_position(ps, 100, 100);
```

```
// Destroys the particle system.  Particle systems are
persistent so it's good practice to destroy particle
systems no longer needed.
part_system_destroy(ps);
```

In the previous code block, a new particle system is created using `part_system_create`. The ID returned can then be used to manipulate the particle system using the other functions introduced.

By default, particle systems are updated and drawn automatically, but they can be configured to either play at different rates or pause entirely.

Emitting particles

Particle emitters can be used to draw particles in a specific region on the screen. The following code sample explains the different functions associated with particle emitters:

```
/// Particle Emitter Code Sample

// Creates a particle emitter associated with the
given system.
pe = part_emitter_create(ps);

// Clears the emitter to its default properties.
part_emitter_clear(ps, pe);

// Returns whether or not the given particle emitter
exists.
part_emitter_exists(ps, pe);

/* Set the position, shape, and distribution of the
particle emitter.
  argument0 is the particle system.
  argument1 is the particle emitter.
  argument2 and argument3 are the x minimum and x
maximum in which particles will be created.
  argument4 and argument5 are the y minimum and y
maximum in which particles will be created.
  argument6 is the shape the emitter takes.
  argument7 is the type of distribution the emitter
uses.
```

```
part_emitter_region(ps, pe, 0, 100, 0, 100,
ps_shape_line, ps_distr_linear);

 Burst a specified number of particles from an
emitter.  Occurs once and is best for one-off effects
such as explosions.
  argument0 is the particle system ID.
  argument1 is the particle emitter ID.
  argument2 is the particle type ID.
  argument3 is the number of particles to burst.
part_emitter_burst(ps, pe, pt, 100);

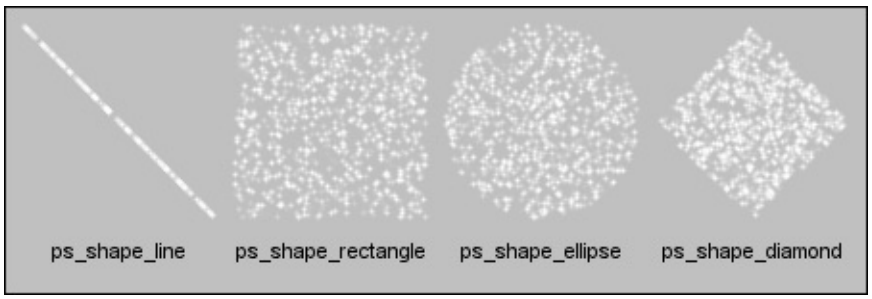
 Streams a specified particle type from an emitter.
Occurs every step and is best for persistent effects
such as rain.
  argument0 is the particle system ID.
  argument1 is the particle emitter ID.
  argument2 is the particle type ID.
  argument3 is the number of particles to be produced
every step. */
part_emitter_stream(ps, pe, pt, 30);

// Destroys the emitter, removing it from memory.
Particles will stop being produced by the emitter but
not cleared.
part_emitter_destroy(ps, pe);

// Destroys all emitters associated with the given
particle system.
part_emitter_destroy_all(ps);
```

Shaping and distributing within emitters

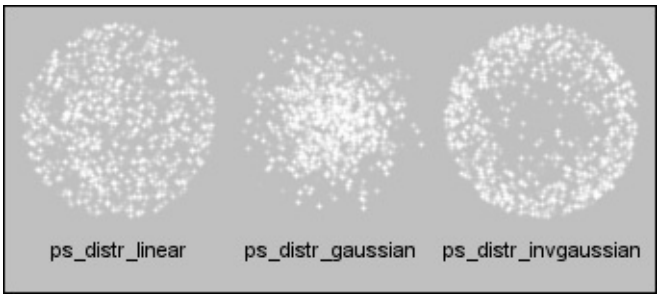
After creating a particle emitter using `part_emitter_create`, it's best to set up the particle emitter's region using `part_emitter_region`. Then, `argument2` through `argument4` define two coordinate points at which the emitter will create particles. The remaining two arguments, however, deal with the shape and distribution of the particles. The four types of shapes and distribution are displayed in the following screenshot:



Then, there are three types of distributions that can be used for `argument7` in `part_emitter_region` as follows:

- `ps_distr_linear`: These particles have an equal chance of appearing anywhere in the region
- `ps_distr_gaussian`: These particles are more likely to be generated around the center of the emitter's region than around the edges
- `ps_distr_invgaussian`: These particles are more likely to be generated around the edges of the emitter's region than the center

The following screenshot showcases how these three distributions look:



Creating different particle types

Particle types define the look and movement of particle effects. Particle types have many different functions to define these properties. The first three functions shown in the following code deal with creating and clearing particles as well as setting the lifetime range of a particle type:

```
// Creates a new particle type, returning its ID.
pt = part_type_create();

// Resets all properties of the specified particle
type to its default.  Does not change or clear
```

```
particles currently being rendered of this type.
part_type_clear(pt);

/* Determines how long a particle type will last in
frames.
  argument0 is the particle type ID.
  argument1 is the minimum amount of life a particle
will exist.
  argument2 is the maximum amount of life a particle
will exist. /
part_type_life(pt, 10, 50);
```

Similar to particle systems, a particle type must first be created before it can be manipulated. This next set of functions deals with the shape, color, and transparency of particle types, as shown in the following code:

```
/* Defines a shape for the particle type to use.
 * argument1 is the shape of the particle, using
several predefined constants.
part_type_shape(pt, pt_shape_circle);

 Defines a sprite for the particle type to use.
  argument1 is the sprite resource ID.
  argument2, if true, the particle will animate.
  argument3, if true, the particle animation will
stretch over the lifetime of the particle.
  argument4, if true, a random subimage of the sprite
will be chose to represent the particle.
/
part_type_sprite(pt, spr_particle, false, false,
false);

// Sets the blending color of the specified particle
type.
part_type_color1(pt, c_red);

/ Sets the blending of two colors over the lifetime of
a particle.
 * The particle will start at as the color specified
as argument1 and interpolate to the color used in
argument2.
part_type_color2(pt, c_red, c_white);

 Sets the blending of three colors over the lifetime
of a particle.
```

```
 * The particle will start as the color specified as
argument1 and interpolate to the color used in
argument2 around 50% of its life, and then finally to
color 3
part_type_color3(pt, c_red, c_white, c_blue);

 Defines two colors that the created particles will
interpolate between
part_type_color_mix(pt, c_black, c_white);

 Defines ranges for the blending color to be for the
specified particle using rgb channels.
  argument1 and argument2 are the min and max range of
the red channel.
  argument3 and argument4 are the min and max range of
the green channel.
 * argument5 and argument6 are the min and max range
of the blue channel.
part_type_color_rgb(pt, 0, 128, 0, 128, 0, 128);

 Defines ranges for the blending color to be for the
specified particle using hue, saturation, and value.
  argument1 and argument2 are the min and max range of
the hue.
  argument2 and argument3 are the min and max range of
the saturation.
  argument4 and argument5 are the min and max range of
the value. /
part_type_color_hsv(pt, 0, 128, 0, 128, 0, 128);

// Defines the alpha or transparency -- 0 to 1 -- of
the specified particle type.
part_type_alpha1(pt, 0.55);

// Defines the alpha values that the specified
particle type will interpolate between during its
lifetime.
part_type_alpha2(pt, 1, 0);

// Defines three alpha values for the specified
particle type to interpolate between during its
lifetime.
part_type_alpha3(pt, 0, 1, 0);

// Defines whether the specified particle type is
rendered using additive blending (true) or alpha
blending (false).part_type_blend(pt, true);
```

There are many different ways to affect the visual appearance of a particle type. There are also many different functions available to animate a particle type, as shown in the following block of code:

```
// Defines the x and y scale of the specified
particle.
part_type_scale(pt, x_scale, y_scale);

/* Defines the range of particle's size.
  argument1 and argument2 define the minimum and
maximum size range.
  argument3 determines how much the size of the
particle should increase (or decrease if negative).
 * argument4 defines a random value to be added to the
particle every step.
part_type_size(pt, 0.1, 1, 0.001, 0);

 Defines the direction that the particle moves.
  argument1 and argument2 define the minimum and
maximum angles of the direction in degrees.
  argument3 will be added to the direction every step.
 * argument4 will specify a random value that will be
added to the direction every frame.
part_type_direction(pt, 0, 360, 0, 0);

 Determines the visual rotation of the particle.
  argument1 and argument2 define the minimum and
maximum angles of the orientation in degrees.
  argument3 will be added to the orientation of the
particle every step.
  argument4 will determine a random value that will be
added to the orientation of the particle every step.
  argument5 is a boolean that determines whether the
particle's orientation will be relative to the angle
of its directional movement.
part_type_orientation(pt, 0, 360, 0, 0, false);

 Determines the speed at which the particle moves.
  argument1 and argument2 define the minimum and
maximum speed of the particle.
  argument3 is the amount of speed that will be added
to the particle every step.
 * argument4 is the amount of randomized speed that
will be added to the particle every step.
part_type_speed(pt, 1, 2, 0, 0);
```

```
 Determines the gravity of the specified particle.
  argument1 is the strength of gravity.
  argument2 is the angle of the gravity in degrees.
part_type_gravity(pt, 1, 270);

 Defines a particle to be created every step of the
current particle's lifetime.
  argument1 is the number of particles to create.
  argument2 is the ID of the other particle to create.
 * NEVER use the same particle type for argument0 and
argument2 as this will cause an infinite loop and
crash the game.
part_type_step(pt, 1, other_pt);

 Defines a particle to be created on the death of the
current particle type.
  argument1 is the number of particles to be created.
  argument2 is the type of particle to be created. */
part_type_death(pt, 100, other_pt);

// Returns whether or not the specified particle type
exists.
part_type_exists(pt)

// Removes the specified particle type from memory.
part_type_destroy(pt);
```

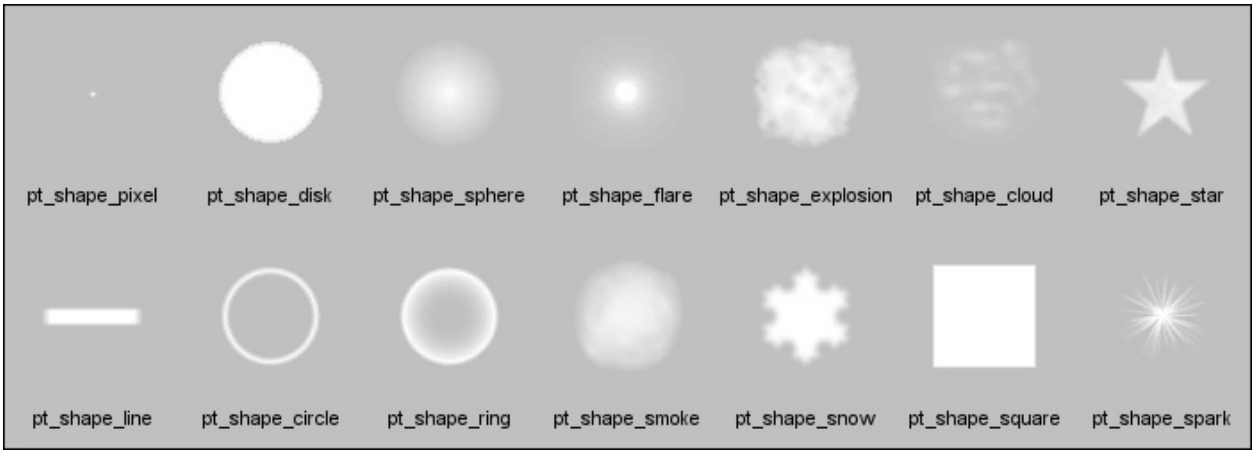
Shaping particle types

Though a sprite resource can be defined for a particle using `part_type_sprite`, GameMaker: Studio provides the following series of particle shapes in the function `part_type_shape` that can be used to define the shape of a particle:

- `pt_shape_pixel` (the default setting)
- `pt_shape_disk`
- `pt_shape_square`
- `pt_shape_line`
- `pt_shape_star`
- `pt_shape_circle`
- `pt_shape_ring`
- `pt_shape_sphere`

- `pt_shape_flare`
- `pt_shape_spark`
- `pt_shape_explosion`
- `pt_shape_cloud`
- `pt_shape_smoke`
- `pt_shape_snow`

The following screenshot illustrates all of these different shapes:



Particle-specific functions

There are four final functions that can be used to create and manipulate particles, all of which are defined and explained in the following code sample:

```
/* Creates a set of particles at the specified
position.  Similar to having an emitter with no width
or height.
  argument0 is the particle system ID.
  argument1 is where the particles will be created on
the x.
  argiment2 is where the particles will be created on
the y.
  argument3 is the particle type ID.
  argument4 is the number of particles to create. /
part_particles_create(ps, 100, 100, pt, 50);

// Identical to part_particles_create, except
argument5 defines a blend color that overrides the
```

```
particles current color.
part_particles_create_color(ps, 100, 100, pt, 50,
c_red);

// Counts and returns the number of particles
currently managed by the specified particle system.
part_particles_count(ps);

// Removes the visual representation of all particles
currently managed by the specified particle system.
part_particles_clear(ps);
```

In the prior block of code, the first two functions mentioned —`part_particles_create` and `part_particles_create_color`—allow the creation of particles at a specific point. These functions are good to use when you want to create particles at a specific point without the use of emitters.

Integrating particles

Now that the basics of particles effects have been explained, the process of integrating them into the puzzle game can begin. In the puzzle game, matching pieces of three, which is one of the main actions, feels a little boring. So, not only will a pop sound be played when pieces are destroyed, but now a particle effect will burst from the center of each piece. Originally, six pieces were made in `spr_pieces`, so six different effects will be produced. The resulting effect will depend on the `image_index` object of the pieces destroyed. To manage the easy creation of particle systems and particle types, a new object resource will be created.

obj_particle_manager

A new object resource named `obj_particle_manager` will be created first. This object will have a **Create** event and a **Destroy** event. It will be referenced globally and will store all of its particle types as well as the number of particles that should be emitted.

The Create event

The **Create** event of `obj_particle_manager` will create a particle system. It will also create a two-dimensional array that will store the IDs of the particle types and the number of particles that should be emitted when that particle type is used. The following is the entire script that will be written into the **Execute Script** component in the **Create** event:

```
/// Initializes variables for the particle systems.

// Creates a reference to the particle manager.
global.ps_manager = id;

// Creates a particle system.
ps = part_system_create();

// The number of particle types.
```

```
part_count = 6;

/* Initializes first particle type and its parameters.
  A semi-transparent, aqua particle that uses the
first subimage of spr_pieces.
  Last for 60 frames exactly and starts two times
larger, reducing by 10% every frame.
  Using the same values for a function that requires a
minimum and maximum, uses just that single value.
  Lastly, only one particle will be emitted when this
is called.
pt[0, 0] = part_type_create();
part_type_life(pt[0, 0], 60, 60);
part_type_sprite(pt[0,0], spr_pieces, false, false,
false);
part_type_color1(pt[0,0], c_aqua);
part_type_alpha1(pt[0,0], 0.75);
part_type_size(pt[0, 0],],], 2, 2, -0.1, 0);
pt[0, 1] = 1;

 Initializes second particle type.
  Creates a red spark particle that last for about 10
to 15 frames.
  Transitions from opaque red to a fully transparent,
lighter red.
  The particle will move in all directions while
varying speed.
  50 particles will be emitted each time this particle
is requested.

pt[1, 0] = part_type_create();
part_type_life(pt[1,0], 10, 15);
part_type_shape(pt[1, 0], pt_shape_spark);
part_type_color2(pt[1, 0], c_red,
make_color_rgb(255,192,192));
part_type_alpha2(pt[1, 0], 1, 0);
part_type_size(pt[1, 0], 0.5, 1.5, 0, 0);
part_type_speed(pt[1,0], 2.5, 15, 0, 0);
part_type_direction(pt[1,0], 0, 360, 0, 0);
pt[1, 1] = 50;

 Initializes third particle type.
  Creates star particles that transition from
transparent and yellow to opaque lime to transparent
green.
  They will burst upwards, but then be pulled down by
gravity.
```

 * The orientation or rotation will also be random and not relative to its speed.

```
pt[2, 0] = part_type_create();
part_type_life(pt[2,0], 10, 40);
part_type_shape(pt[2, 0], pt_shape_star);
part_type_color3(pt[2, 0], c_yellow, c_lime, c_green);
part_type_alpha3(pt[2, 0], 0, 1, 0);
part_type_speed(pt[2,0], 10, 20, 0, 0);
part_type_direction(pt[2,0], 45, 135, 0, 0);
part_type_orientation(pt[2,0], 0, 360, 0, 0, false);
part_type_gravity(pt[2,0], 2, 270);
pt[2, 1] = 10;
```

 Initializes fourth particle type.
  Line particles that burst outward from their center point, oriented by their direction.
  The colors transition from black to yellow but is also additive.
 * The consistent speed will be slowed down every step, giving the illusion of friction.

```
pt[3, 0] = part_type_create();
part_type_life(pt[3,0], 10, 15);
part_type_color_mix(pt[3,0], c_black, c_yellow);
part_type_alpha2(pt[3,0], 1, 0);
part_type_blend(pt[3,0], true);
part_type_shape(pt[3,0], pt_shape_line);
part_type_speed(pt[3,0], 15, 15, -1, 0);
part_type_direction(pt[3,0], 0, 360, 0, 0);
part_type_orientation(pt[3,0], 0, 0, 0, 0, true);
pt[3, 1] = 10;
```

 Initializes fifth particle type.
  Creates rings that are squashed on their y axis due to the adjustment in scale.
  They transition from fully opaque aqua to semi transparent white and ending in fully transparent blue.
 * They are initially launched downwards and pulled up by their gravity and spin.

```
pt[4, 0] = part_type_create();
part_type_life(pt[4,0], 15, 45);
part_type_shape(pt[4,0], pt_shape_circle);
part_type_scale(pt[4,0], 1, 0.5);
part_type_color3(pt[4,0], c_aqua, c_white, c_blue);
part_type_alpha3(pt[4,0], 1, 0.75, 0);
part_type_speed(pt[4,0], 4, 15, -0.015, 1);
```

```
part_type_direction(pt[4,0], -135, -45,0,0);
part_type_orientation(pt[4,0],45, 135, 15, 0, false);
part_type_gravity(pt[4,0], 1, 90);
pt[4, 1] = 10;

 Initializes sixth particle type.
  Creates disk-shaped particles that grow using
wiggle.
  They are shot out and also reduced over time.
  Their colors are set using randomized hues,
saturations, and values.
  They also rotated around their origin. */
pt[5, 0] = part_type_create();
part_type_life(pt[5,0], 15, 30);
part_type_shape(pt[5, 0], pt_shape_disk);
part_type_color_hsv(pt[5, 0], 192, 255, 0, 255, 128,
255);
part_type_alpha2(pt[5, 0], 1, 0);
part_type_size(pt[5,0], 0.25, 0.35, 0, 1);
part_type_direction(pt[5,0], 0, 360, 12.5, 0);
part_type_speed(pt[5, 0], 2, 4, -0.05, 0);
pt[5, 1] = 10;
```

The code for this **Create** event appears to be rather lengthy, but it's really just establishing six different particle types.

The first part of the **Create** event sets up some parameters. Initially, a global reference to the ID of `obj_particle_manager` is made in a similar manner to the one in `obj_grid_manager`. Then, the particle system, which is an instanced variable, is created. Afterwards, the number of particle types that will be created is tracked. This will be explained when the **Destroy** event is created.

It's also important to note that `pt` is a two-dimensional array. At `pt[n, 0]` where `n` represents the `image_index` object of the destroyed puzzle piece, the ID of the particle type is stored. Then, at `pt[n, 1]`, the number of particles that will be emitted when calling `part_particles_create` is stored. Because the point at which particles will burst is just a single point, emitters are not needed and have not been created.

The Destroy event

The job of the **Destroy** event of `obj_particle_manager` is to destroy each particle type created in this game as well as the particle system, removing them from memory, which will in turn prevent memory-related issues. The code for this event is as follows:

```
///Destroys all particle types managed by this object.

// Destroys each particle type first.
var i;
for (i = 0; i < part_count; i++)
{
    part_type_destroy(pt[i, 0]);
    pt[i, 0] = noone;
}

// Destroys the particle system.
part_system_destroy(ps);
ps = noone;

// Removes the reference of this object from the
global variable.
global.ps_manager = noone;
```

In the previous block of code, observe that `part_count` is used, which was initialized in the **Create** event. First, each particle type is destroyed. Then, the particle system itself is destroyed. Finally, the global reference to the destroyed instance of `obj_particle_manager` is set to `noone`.

Placing obj_particle_manager

To utilize `obj_particle_manager`, an instance of it can be created using `instance_create` in the **Create** event of an object, but it can also just be placed in the `rm_gameplay` room, which will be made in this game. So, `rm_gameplay` should now have two objects, both represented by blue circles with red question marks inside, since they do not have sprites associated with them.

Creating bursts within scr_reorganize_board

Now that `obj_particle_manager` is placed in `rm_gameplay`, this object can be called in scripts. This will be done in `scr_reorganize_board` in the first pair of the `for` loops after determining that a piece has been matched and is going to be "destroyed", as shown in the following code:

```
if (array_pieces[i, j].matched)
{
  // The local array is increased and fill it with the
pieces   that are matching.
  array_broken_piece[array_length++] = array_pieces[i,
j];

   // Emits particles from the position of the piece
and itsindex.
  var burst_x, burst_y, index;
  burst_x = array_pieces[i, j].x;
  burst_y = array_pieces[i, j].y;
  index = array_pieces[i, j].image_index;
  with (global.ps_manager)
  {
    part_particles_create(ps, burst_x, burst_y,
      pt[index,0], pt[index,1]);
  }
}
```

So, after storing the reference of the matching piece, its `x` and `y` positions and the `image_index` values are all stored as local variables. Then, using a `with` statement that references the particle manager using `global.ps_manager`, these local variables are utilized and the appropriate particle type is created. Again, the previous argument of `part_particles_create` now gives the number of particles to create, which is defined within the two-dimensional array that also contains the IDs of each particle type.

If placed properly, when playing the game, a corresponding particle type should be created every time the player makes a matching set. This not only gives the puzzle game more feedback, but also makes that feedback juicier, more entertaining, and more rewarding for the player.

Summary

In this chapter, the puzzle game that was started in Chapter 2, *Random Organization – Creating a Puzzle Game*, now gives feedback in three different ways. Alarms were created and integrated so that pieces can be moved and there are delays in their breaks instead of it all happening instantaneously.

Then, audio was discussed. Sound effects were added to the project and used for various kinds of feedback. Music was also added to the game and faded in to eliminate complete silence in the game.

Finally, particle effects were discussed. Six particle types were defined in a new object resource that was integrated into the game giving extra feedback and flair.

The puzzle game should now be playable with improved feedback, but a few things are still missing. Though the game has rules such as the player only being able to swap pieces to make a matching set of three, there is no "win-or-lose" condition and no score. It feels more like a toy than a game. Also, only four of the six pieces and particle types created are currently used in the game.

In the next chapter, features that will make the puzzle game feel more complete will be added. These include scoring and the UI, a lose condition, and a game over state through the use of a timer. Also, a menu to start the game in and return to after a gameplay session where various aspects of the puzzle game can be changed, such as the size of the board and the number of piece types to use will be made. The next chapter will be the final one that discusses the puzzle game before we move on to a new project.

Chapter 5. Solving the Puzzle – Finishing Touches to the Puzzle Game

In the previous chapter, effects were added to the puzzle game in the hope of adding a little bit of flare, but more importantly, player feedback. Right now, the puzzle game is in a playable state, but a few elements are missing. Firstly, no parameters about the game can be adjusted. The game always uses 10 columns and 6 rows. It would be nice if the player could change this. Secondly, the addition of particle bursts is nice, but the player needs a cumulative reward they can see, something to show they are making progress. Finally, there is no losing condition or pressure to perform, which can result in a rather boring experience.

In this chapter, the following additions will address these issues and give the puzzle game a more complete feel:

- A main menu will be rendered using the **Draw** events and GameMaker: Studio's built-in `draw` functions
- Functionality will be scripted into this menu, allowing the player to adjust the number of rows, columns, pieces used, and the length of time a play session will last
- The player will have their score and the time remaining for the play session to end tracked and displayed, creating time pressure
- Once the timer runs out, the player will enter a brief "game over" state and then return to the menu allowing them to play the game again

Drawing and graphical user interface

Before jumping into discussing the construction of the main menu,

GameMaker: Studio's **Draw** and **Draw GUI** events and the built-in functions associated with them will be expanded upon.

In GameMaker: Studio, there are several dozen built-in methods that are prefaced by `draw_`. These functions handle drawing a variety of game elements onscreen and will only work if executed during a **Draw** or **Draw GUI** event; these two events are very similar but not identical.

The **Draw** events will draw objects in the game space. If the game view is translated, rotated, or scaled, anything rendered using the **Draw** events is changed accordingly. Also, even if the objects that utilize the **Draw** event have a sprite assigned to them, they will not be rendered automatically. Finally, these elements are bound to the depth of the object that is drawing them.

The **Draw GUI** events render images in direct relation to the game's window, so if the view within a room is transformed, the **Draw GUI** events will remain the same. Also, an object's sprites will still be rendered automatically. Finally, the **Draw GUI** events are always drawn on top of all other game elements, even those rendered during the **Draw** events; this makes the **Draw GUI** event great for drawing interface elements such as score or health bars or any elements that should always be visible while playing.

Understanding draw functions

There are many different `draw` functions present in GameMaker: Studio. In this section, a good number of the most common functions, such as setting the color in which to draw or rendering primitive shapes, will be covered.

Drawing rectangles

The rectangle is a very common shape drawn in user interfaces and can also be used for simple buttons and health bars. The following code example shows some built-in functions that can be used to draw rectangles and other rectangular shapes:

```
// Draws a rectangle
// argument0 is the first x position.
// argument1 is the first y position.
// argument2 is the second x position.
// argument3 is the second y position.
// argument4 determines if the drawn rectangles is a
one pixel outline (true) or a solid (false).
draw_rectangle(10, 10, 110, 110, true);

// Draws a rectangle, but with specified colors.
// argument0 to argument3 are identical to
draw_rectangle.
// argument4 colors the point at the first x and y.
// argument5 colors the point at the second x and
first y.
// argument6 colors the point at the second x and
second y.
// argument7 colors the point at the first x and
second y.
// argument8 determines if the drawn rectangle is a
one pixel outline (true) or a solid (false).
draw_rectangle_color(120, 10, 220, 110, c_white,
c_black, c_gray, c_ltgray, false);

// Draws a rectangle with rounded edges.  Its
parameters are identical to draw_rectangle.
draw_roundrect(10, 120, 110, 220, true);

// Draws a rectangle with rounded edges using
specified colors.
// argument0 to argument3 are identical to
draw_rectangle.
// argument4 is the center color
// argument5 is the outer color
// argument6 determines if the drawn rectangle is a
one pixel outline (true) or a solid (false).
draw_roundrect_color(120, 120, 220, 220, c_white,
c_black, false);

// Draws a simple, rectangular button.
// argument0 to argument3 are the same as
draw_rectangle.
// argument4 is a boolean that determines how the
button will look.
// If true, the button will look like it is up; if
false, the button will look like it is being pressed
```

```
down.
draw_button(10, 230, 110, 290, true);
draw_button(120, 230, 220, 290, false);

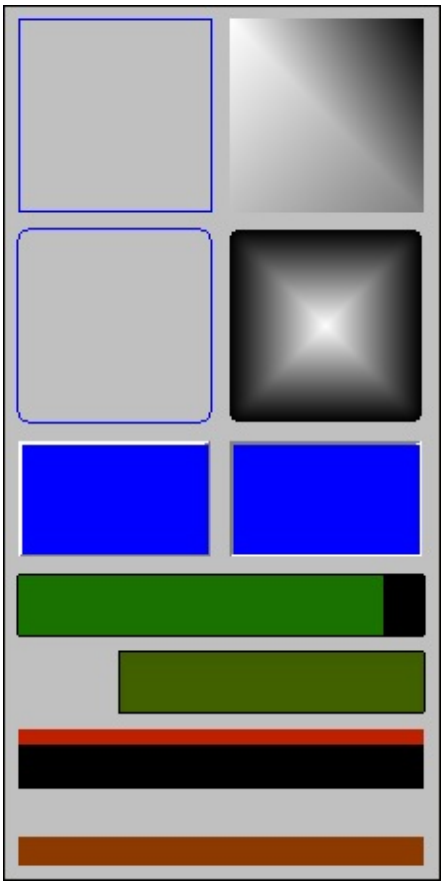
// Draws a health bar based on a specified percent.
// argument0 to argument3 are identical to
draw_rectangle
// argument4 is the amount of the bar if full.  0 is
empty, 100 is full.
// argument5 is the back color of the bar.
// argument6 is the color drawn when the health bar is
empty.
// argument7 is the color drawn when the health bar is
full.
// argument8 is the direction the bar fills.  0 fills
left to right, 1 fills right to left, 2 fills top to
bottom, 3 fills bottom to top.
// argument9 uses argument5's color to draw a backdrop
if true.
// argument10 draws a 1 pixel black border if true.
draw_healthbar(10, 300, 220, 330, 90, c_black, c_red,
c_green, 0, true, true);

// Variation at 75% full without background and moving
right to left
draw_healthbar(10, 340, 220, 370, 75, c_black, c_red,
c_green, 1, false, true);

// Variation at 25% full without border and moving top
to bottom
draw_healthbar(10, 380, 220, 410, 25, c_black, c_red,
c_green, 2, true, false);

// Variation at 45% full without border or background
and moving bottom to top.
draw_healthbar(10, 420, 220, 450, 45, c_black, c_red,
c_green, 3, false, false);
```

The following screenshot shows what the preceding code sample will generate:



Setting color and alpha

Some `draw` functions do not request a color as an argument, such as `draw_rectangle`. To set the color and alpha values when drawing various shapes, one of the following `set` functions can be executed before executing a `draw` function:

- `draw_set_color(color)`: This function sets the color that will be used by all of the `draw` functions that do not take colors as arguments
- `draw_set_alpha(alpha)`: This function sets the transparency, ranging from `0.0` to `1.0`, of the `draw` functions that do take an `alpha` value as an argument.

Note

Setting a `draw` parameter is a persistent action and will continue to be

used across all of the **Draw** events. For example, if one object sets the draw color to `c_red`, then all following `draw` functions that are executed, even from other objects, will use `c_red` as their draw color value. It is a good habit to keep track of and set these parameters often when using **Draw** events.

Drawing circles and ellipses

Circles and ellipses are also important shapes and are useful to draw targets or highlight a point on a map interface. The following code sample shows several functions used to draw circles and ellipses:

```
// Draws a circle
// argument0 is the horizontal center of the circle.
// argument1 is the vertical center of the circle.
// argument2 is the circle's radius.
// argument3 determines if the circle will be a one
pixel outline (true) or a solid (false).
draw_circle(50, 50, 40, true);

// Draws a circle but with specified colors.
// argument0 to argument2 are the same as draw_circle.
// argument3 is the color at the center of the circle.
// argument4 is the color at the outer edge of the
circle.
// argument5 is the same as argument3 in draw_circle.
draw_circle_color(150, 50, 40, c_black, c_white,
false);

// Draws an ellipse.
// argument0 is the first x position.
// argument1 is the first y position.
// argument2 is the second x position.
// argument3 is the second y position.
// argument4 determines if the drawn circle will be an
outline (true) or solid (false).
draw_ellipse(10, 100, 200, 200, true);

// Draws an ellipse, but with specified colors.
// argument0 to argument3 are identical to
draw_ellipse
// argument4 the center color of the ellipse.
```

```
// argument5 the outer color of the ellipse.
// argument6 is the same as argument4 in draw_ellipse.
draw_ellipse_color(210, 100, 390, 200, c_white,
c_black, false);
```

Setting a circle's precision

Along with color and alpha, circles also have a `precision` parameter, which can be set with `draw_set_circle_precision(precision)`. The value assigned to `argument0` in this function must satisfy two rules to be allowed:

- The value must fall within the range of 4 and 64
- The value must be divisible by 4

The following code sample shows how to draw circles with different precision values:

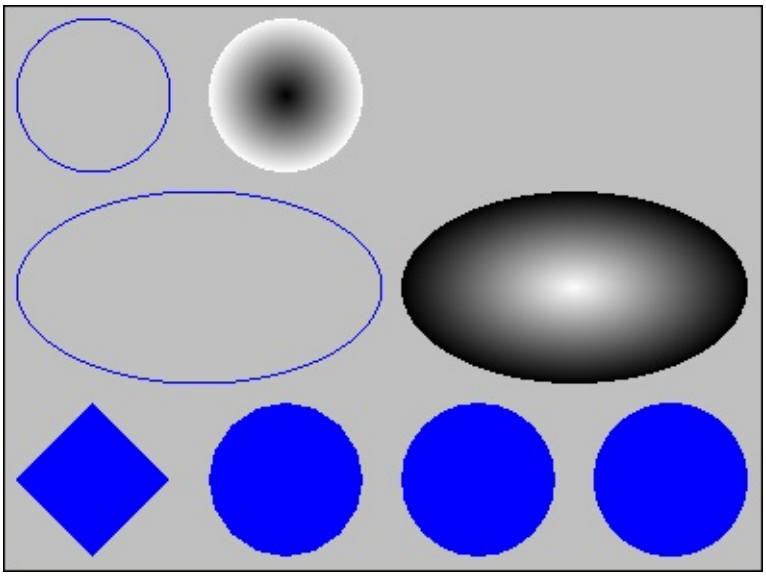
```
// Lowest precision, 4.
draw_set_circle_precision(4);
draw_circle(50, 250, 40, false);

// Precision of 16
draw_set_circle_precision(16);
draw_circle(150, 250, 40, false);

// Default precision, 24
draw_set_circle_precision(24);
draw_circle(250, 250, 40, false);

// Highest precision
draw_set_circle_precision(64);
draw_circle(350, 250, 40, false);
```

The following screenshot shows circles and ellipses drawn using the previous two code samples:



Drawing points, lines, and arrows

Drawing points, lines, and arrows can sometimes be very useful as visual hints as well as for debugging purposes. The following code sample shows functions used to draw lines and arrows:

```
// Draws a point on screen.
// argument0 is the x coordinate.
// argument1 is the y coordinate.
draw_point(50, 50);

// Draws a point with a specified color.
// argument0 and argument1 reflect draw_point.
// argument2 is the color the point will be.
draw_point_color(75, 50, c_white);

// Draws a one-pixel wide line.
// argument0 is the starting x point of the line.
// argument1 is the starting y point of the line.
// argument2 is the ending x point of the line.
// argument3 is the ending y point of the line.
draw_line(100, 10, 100, 110);

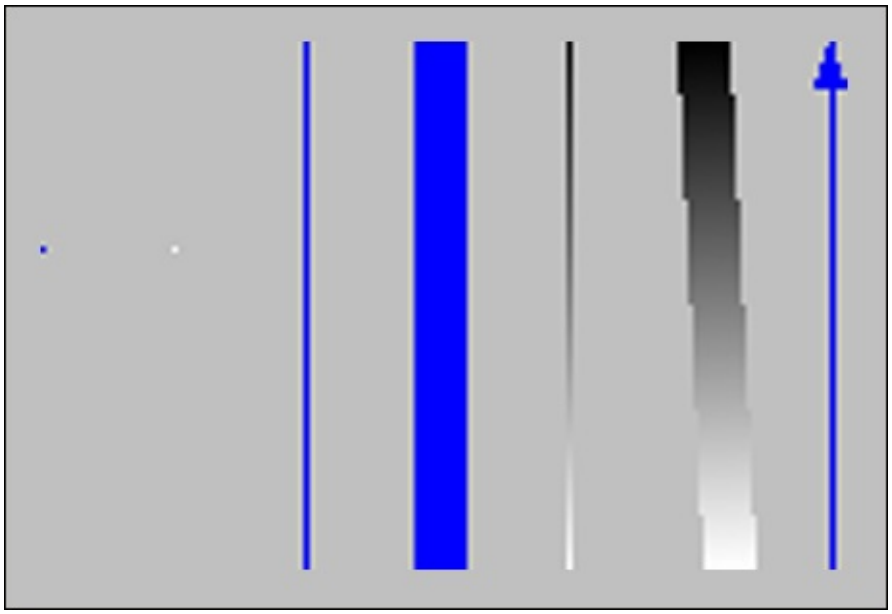
// Draws a line with a specified width.
// argument0 to argument3 reflects those in draw_line.
// argument4 is the width in pixels of the line.
draw_line_width(125, 10, 125, 110, 10);
```

```
// Draws a line with specified colors.
// argument0 to argument3 reflects those in draw_line.
// argument4 is the color of the line at the start
point.
// argument5 is the color of the line at the end
point.
draw_line_color(150, 10, 150, 110, c_black, c_white);

// Draws a line with specified colors and width.
// argument0 to argument3 reflects those in draw_line.
// argument4 is the width in pixels of the line.
// argument6 is the color of the line at the start
point.
// argument7 is the color of the line at the end
point.
draw_line_width_color(175, 10, 180, 110, 10, c_black,
c_white);

// Draws a line with an arrowhead at the end.
// argument0 to argument3 are similar to those in
draw_line.
// argument4 is the length of the arrowhead.
// The arrowhead will be drawn and angled properly at
the end point with a length proportional to the
line's.
draw_arrow(200, 110, 200, 10, 10);
```

The following screenshot shows the result of the functions described in the previous code; it is scaled to show them in more detail as lines and points are only 1 pixel in size by default:



Drawing text

Drawing text is a very common action for **Draw** events. It can be used to display the score, a character's name, messages, and a variety of other important information. There are many different ways to display text in GameMaker: Studio, as demonstrated by the following code sample:

```
// Draws text.
// argument0 is the x coordinate.
// argument1 is the y coordinate.
// argument2 is the text that will be drawn.
draw_text(100, 50, "Text");

// Draws the text with extended parameters.
// argument0 to argument2 are identical to draw_text.
// argument3 is the spacing between lines of text if
there are more than one.
// argument4 is a width that, if exceeded by the text,
new lines will automatically be created.
draw_text_ext(100, 100, "My Extended Text", 20, 50);

// Draws text while transforming it.
// argument0 to argument2 are identical to draw_text.
// argument3 is the horizontal scale of the text.
// argument4 is the vertical scale of the text.
// argument5 is the angle of the text.
draw_text_transformed(100, 165, "Transformed Text",
```

```
1.1, 1.3, 10);

// Draws the text with specified colors.
// argument0 to argument2 are identical to draw_text.
// argument3 is the upper left color.
// argument4 is the upper right color.
// argument5 is the bottom right color.
// argument6 is the bottom left color.
// argument7 is the transparency (0 to 1) of the text.
draw_text_color(100, 200, "Colored Text", c_black,
c_red, c_blue, c_red, 1.0);

// Renders text using a combination of draw_text_ext
and draw_text_transformed.
// argument0 to argument4 are the same as
draw_text_ext.
// argument5 to argument7 are the same as
draw_text_transformed.
draw_text_ext_transformed(100, 300, "Extended,
Transformed Text", 15, 100, 1.1, 1.2, -10);

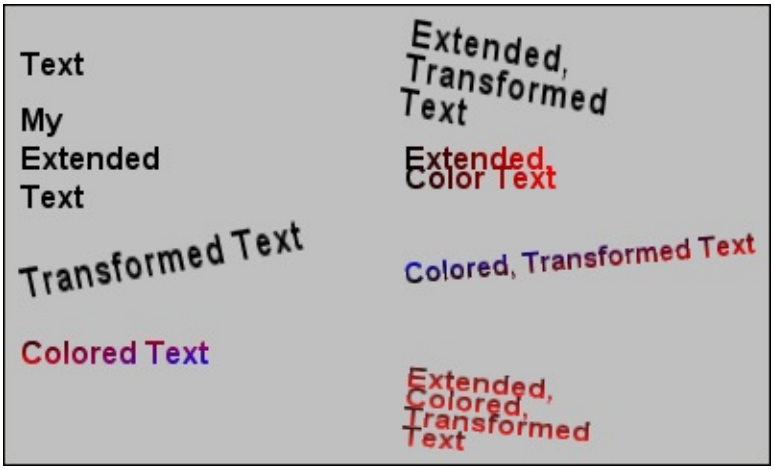
// Draws the text using a combination of draw_text_ext
and draw_text_color.
// argument0 to argument4 are the same as
draw_text_ext.
// argument5 to argument9 are the same as
draw_text_color.
draw_text_ext_color(100, 235, "Extended, Color Text",
10, 100, c_black, c_red, c_red, c_black, 1.0);

// Draws text using a combination of
draw_text_transformed and draw_text_color.
// argument0 to argument5 are the same as
draw_text_transformed.
// argument6 to argument10 are the same as
draw_text_color.
draw_text_transformed_color(100, 360, "Colored,
Transformed Text", 0.9, 0.9, 5, c_blue, c_black,
c_red, c_black, 1.0);

// Draws text using a combination of all three text
functions.
// argument0 to argument7 are the same as
draw_text_ext_transformed.
// argument8 to argument12 are the same as
draw_text_color.
draw_text_ext_transformed_color(100, 405, "Extended,
```

```
Colored, Transformed Text", 12, 50, 1.0, 0.85, -5,
c_black, c_black, c_red, c_red, 0.9);
```

The preceding code sample will create lines of text, as illustrated in the following screenshot:



Setting font and text alignment

When drawing text, three functions exist that set font and alignment:

- `draw_set_font(font_resource_id)`: This function sets the font of the text
- `draw_set_halign(horizontal_alignment)`: This function horizontally aligns the text
- `draw_set_valign(vertical_alignment)`: This function vertically aligns the text

There are six constants that can be used for the alignment of text on the screen: three for horizontal alignment and three for vertical alignment. They are as follows:

- `fa_left`: This constant is used to horizontally align the text to the left
- `fa_center`: This constant is used to horizontally align the text to the center
- `fa_right`: This constant is used to horizontally align the text to the right
- `fa_top`: This constant is used to vertically align the text to the top

- `fa_middle`: This constant is used to vertically align the text to the center
- `fa_bottom`: This constant is used to vertically align the text to the bottom

Drawing sprites

The previous functions have demonstrated the rendering of primitive shapes or text; however, sprites can also be drawn in a variety of ways using `draw` functions, as demonstrated in the following code sample and the accompanying screenshot:

```
// Draws the current object at its current location,
using its current sprite.
draw_self();

// Draws the specified sprite.
// argument0 is the sprite resource id.
// argument1 is the sub-image.
// argument2 is the x position.
// argument3 is the y position.
draw_sprite(spr_pieces, 0, 200, 100);

// Draws a portion of a specified sprite as if
cropping it.
// argument0 and argument1 are the same as
draw_sprite.
// argument2 is how many pixels in from the left are
cropped.
// argument3 is how many pixels in from the top are
cropped.
// argument4 is the width of the drawn portion of the
sprite.
// argument5 is the height of the drawn portion of the
sprite.
// argument6 is the x position.
// argument7 is the y position.
draw_sprite_part(spr_pieces, 4, 12, 12, 40, 40, 300,
50);

// Draws a sprite, stretched to fit within a specified
position and size.
// argument0 and argument1 are identical to those in
```

```
draw_sprite.
// argument2 is the x position.
// argument3 is the y position.
// argument4 is the width in which the sprite will be
stretched to match.
// argument5 is the height in which the sprite will be
stretched to match.
draw_sprite_stretched(spr_pieces, 5, 400, 25, 50, 88);

// Draws a sprite with extended parameters.
// argument0 to argument3 are the same as draw_sprite.
// argument4 is the x scale.
// argument5 is the y scale.
// argument6 is the angle in degrees.
// argument7 is the blend color.
// argument8 is the alpha.
draw_sprite_ext(spr_pieces, 2, 100, 200, 0.5, 1.1,
12.5, c_ltgray, 0.9);

// Renders a sprite using a combination of
draw_sprite_part and draw_sprite_ext.
// argument0 to argument7 are identical to
draw_sprite_part.
// argument8 is the x scale.
// argument9 is the y scale.
// argument10 is the color.
// argument11 is the alpha.
draw_sprite_part_ext(spr_pieces, 4, 0, 0, 55, 75, 200,
200, 0.9, 1, c_gray, 1.0);

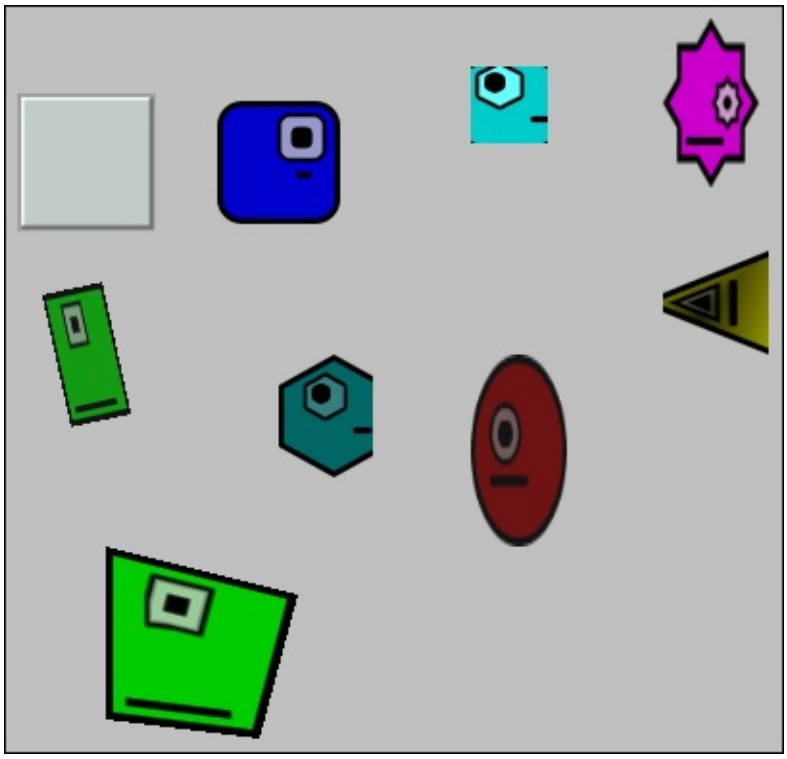
// Renders a sprite using a combination of
draw_sprite_stretched and draw_sprite_ext.
// argument0 to argument5 are identical to
draw_sprite_stretched.
// argument6 is the blend color.
// argument7 is the alpha.
draw_sprite_stretched_ext(spr_pieces, 1, 300, 200, 50,
100, c_gray, 0.9);

// Draws a sprite with the most number of available
options.
// argument0 to argument7 reflect those in
draw_sprite_part.
// argument8 is the x scale.
// argument9 is the y scale.
// argument10 is the angle in degrees.
// argument11 is the upper left blend color.
```

```
// argument12 is the upper right blend color.
// argument13 is the lower right blend color.
// argument14 is the lower left blend color.
// argument15 is the alpha of the drawn sprite.
draw_sprite_general(spr_pieces, 3, 5, 10, 55, 44, 400,
200, 1.0, 1.25, 90, c_black, c_white, c_gray,
c_ltgray, 1.0);

// Draws a sprite with each corner at a specified
coordinate.  Great for skewing a sprite.
// argument0 and argument1 are the same as
draw_sprite.
// argument2 is the first x position.
// argument3 is the first y position.
// argument4 is the second x position.
// argument5 is the second y position.
// argument6 is the third x position.
// argument7 is the third y position.
// argument8 is the fourth x position.
// argument9 is the fourth y position.
// argument10 is the alpha transparency.
draw_sprite_pos(spr_pieces, 2, 110, 300, 210, 325,
190, 400, 110, 390, 0.75);
```

The following screenshot is what should result from running the previous code. Also, the current object is assigned `spr_grid` as its sprite and is located at the position `100` on both the `x` and `y` coordinates:



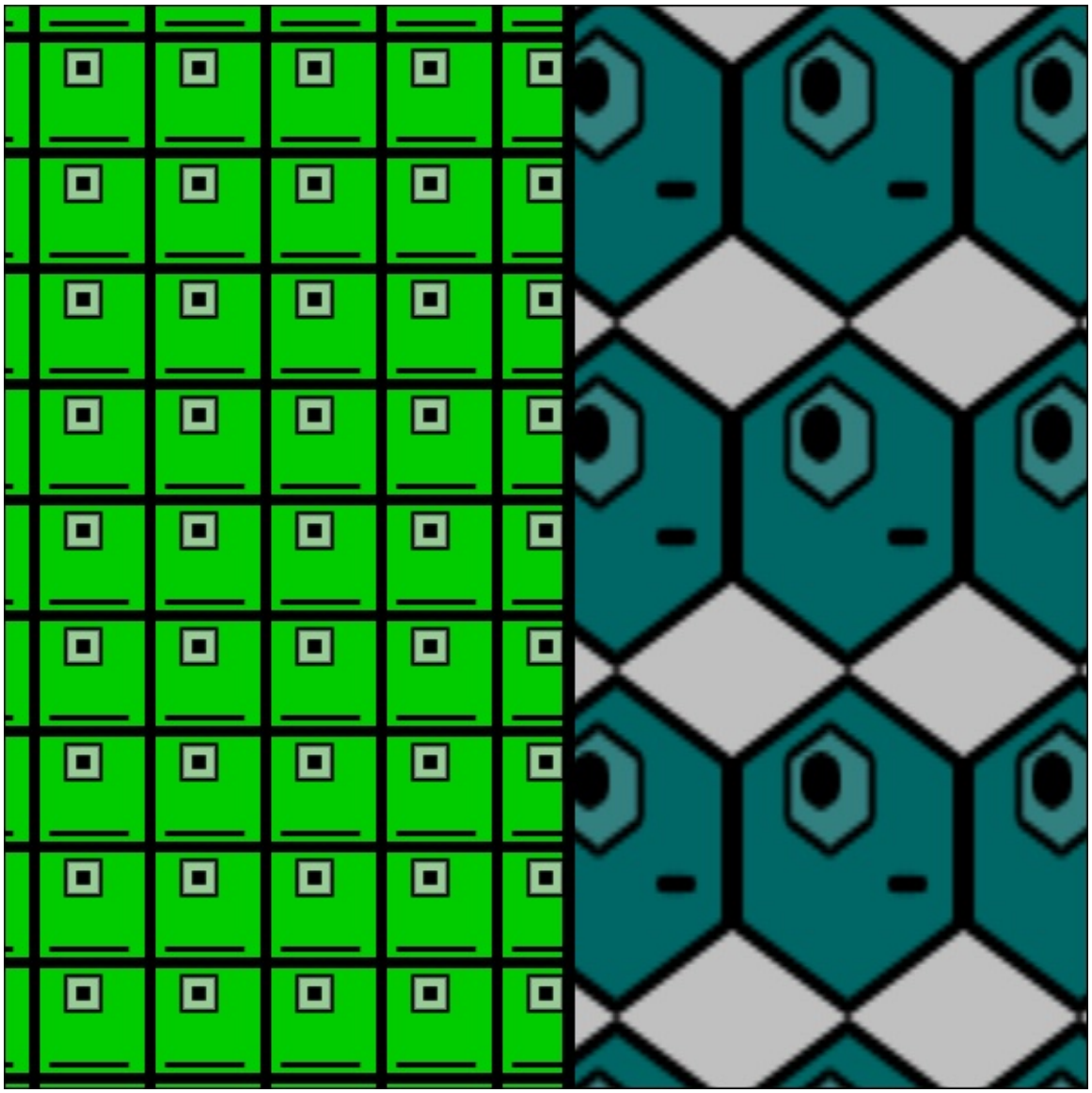
Tiling sprites

Sprites can also be tiled across the entire screen using the two functions demonstrated in the following code sample:

```
// Draws sprite tiling across the entire screen.
// argument0 is the sprite index.
// argument1 is the sub-image.
// argument2 is the x offset.
// argument3 is the y offset.
draw_sprite_tiled(spr_pieces, 2, 50, 50);

// Draws a sprite tiling across the entire screen with
added parameters.
// argument0 to argument3 are the same as
draw_sprite_tiled.
// argument4 is the x scale.
// argument5 is the y scale.
// argument6 is the blend color.
// argument7 is the alpha.
draw_sprite_tiled_ext(spr_pieces, 4, 0, 0, 0.5, 0.25,
c_gray, 1.0);
```

The next screenshot demonstrates the result of executing the two functions in the previous code: the green puzzle pieces are tiled to the left and the stretched, slightly darkened cyan pieces are tiled to the right:



Establishing the drawing order

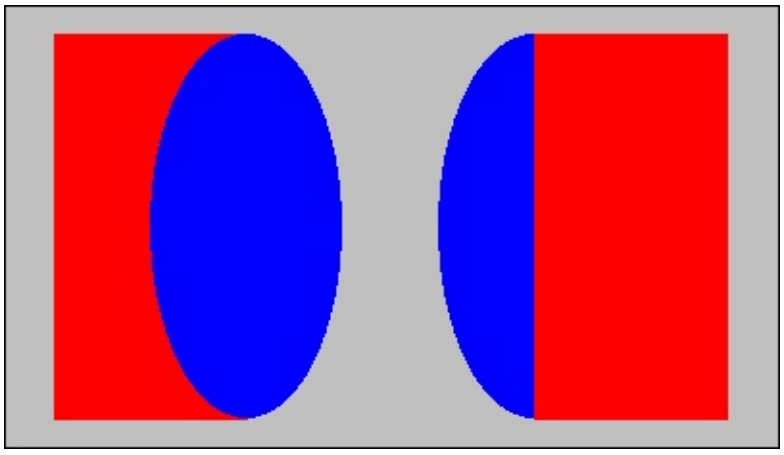
Normally, the order in which objects are rendered is determined by their individual depths. Objects with higher depth values are drawn first as items with lower depths are rendered on top. Within a **Draw** or **Draw GUI** event, this depth is determined by the order in which the `draw` functions are called. Simply put, when an object is drawn, it will be drawn over the

previously drawn object, the next object drawn will be over the current one, and so on. The following code sample draws a blue ellipse and a red rectangle in different orders to demonstrate this concept:

```
// Draws a red rectangle with a blue ellipse above it.
draw_set_color(c_red);
draw_rectangle(0, 0, 100, 200, false);
draw_set_color(c_blue);
draw_ellipse(50, 0, 150, 200, false);

// Draws a blue ellipse with a red rectangle on top of
it.
draw_set_color(c_blue);
draw_ellipse(300, 0, 400, 200, false);
draw_set_color(c_red);
draw_rectangle(350, 0, 450, 200, false);
```

As shown in the following screenshot, to the left of the screenshot, the blue ellipse should be drawn atop the red rectangle and the opposite should happen on the right:



Gathering resources for creating the main menu

Now that the **Draw** and **Draw GUI** events have been covered, we can cover the assets that will need to be created to create the main menu. The following are the assets required to create the main menu:

- `obj_main_menu`: This is the object resource for the main menu
- `rm_main_menu`: This is the room that will contain an instance of `obj_main_menu`
- `scr_on_menu_button_pressed`: This is the tenth script resource, used by `obj_main_menu`
- `fnt_main` and `fnt_title`: These are the two font resources used to render text

Creating obj_main_menu

The main menu object resource will allow players to edit parameters, such as the number of columns and rows, before starting the game. All of this information can be changed using mouse and keyboard input and rendered using the **Draw** events. This object will have no sprite assigned to it, but will require the following events:

- The **Create** event, in which many global and instanced variables will be defined
- The **Draw** event, in which all elements of the main menu will be rendered
- The **Global Left Button** and **Global Left Released** events will handle input from the mouse
- The **press <any key>** and **release <any key>** events will handle input from the keyboard

Building a new room – rm_main_menu

The `rm_main_menu` room will contain the only instance of `obj_main_menu`. This room is of the same size as `rm_gameplay`: 960 pixels by 540 pixels. The `rm_main_menu` room should be the first room the player enters when playing the game. To ensure this, drag-and-drop `rm_main_menu` in the `Rooms` folder of the resource manager so that it is above `rm_gameplay`.

After creating `rm_main_menu`, drop one instance of `obj_main_menu` into this room. Remember, an object resource that is not assigned a sprite resource, such as `obj_main_menu`, will show up as a red question mark with a blue circle behind it.

Creating fonts – fnt_title and fnt_main

Font resources are used to convert fonts into a format that GameMaker: Studio can use, allowing it to render text.

To finish the puzzle game, the following two fonts will be created:

- `fnt_main`: This font is used for most of the text in the game
- `fnt_title`: This font is used for large text

When adding a new font, navigate to **Resources** | **Create Font** or press *Shift* + *Ctrl* + *F*; a **Font Properties** dialog window will pop up. An example window is shown in the following screenshot:



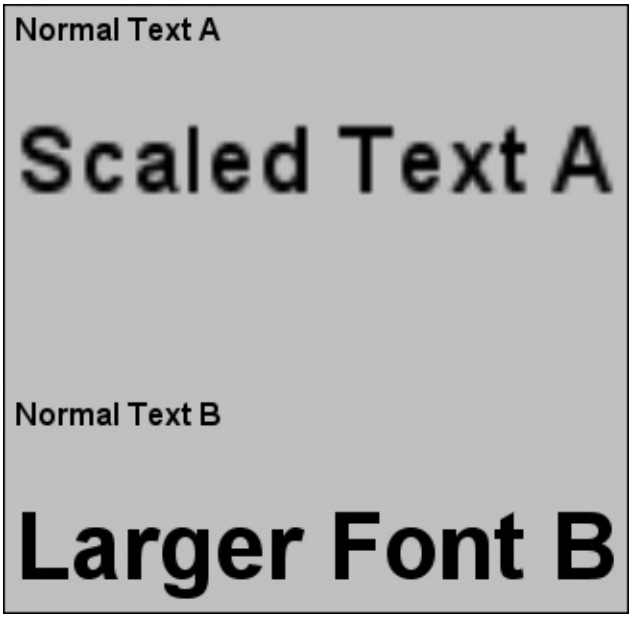
The first parameter on this screen is the font resource's **Name** value,

which is used when setting the current font with `draw_set_font`. Beneath **Name**, a font can be selected from the list of fonts installed on the current computer.

> ## Note
>
> GameMaker: Studio converts and fits fonts to an image, which means there is no need to worry if a user on a different machine does not have the selected font installed.

In the **Style** section, **Anti-Aliasing** can be set from the value **Off** (or **0**) to **3**. The higher the value, the smoother the text's edges will look when it is rendered. Then, the **Size** value of the font can be set.

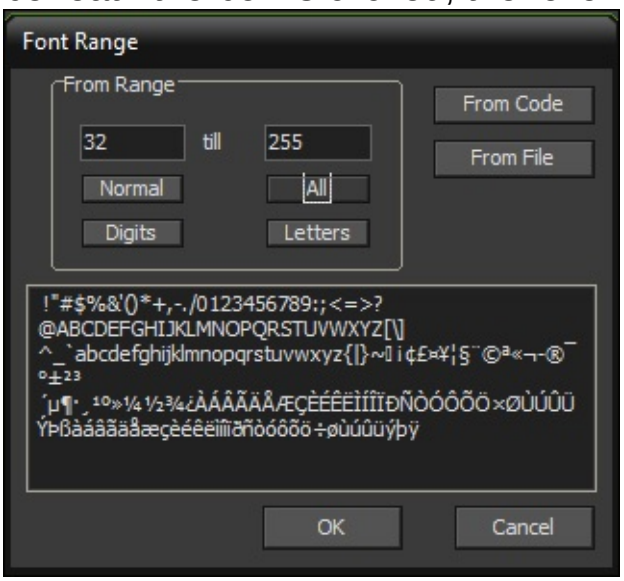
Text can be scaled using `draw_text_transformed` and its variants, but doing so, especially while enlarging the image, can create unpleasant artifacts on the drawn text, as demonstrated in the following screenshot:



Because of these artifacts, it is usually best to create a larger font resource, even if it is of the same font type. Beneath **Size**, there are two checkboxes that can give the resulting font resource a bold look, an italicized look, or both.

The topmost textbox to the right of the **Font Properties** window is for testing and previewing the resulting text. Initially, **Hello World!!** is shown in this preview textbox. Clicking on it will allow any test text to be entered.

The last two boxes represent and help define the font range or characters that can be used with this font resource. The textbox to the left of the screenshot displays all of this text with the resulting font. If the **+** button beneath the box is clicked, the following font range dialog pops up:



The previous dialog can be used to define the characters that are drawn by this font. The numbers displayed are the minimum and maximum ASCII character codes used in the game. For example, **32** represents the exclamation mark. There are six buttons available to make setting up this range easier, as shown in the following list:

- **Normal** (**32 till 127**): These are the most common characters and are usually the best setting for a typical game
- **Digits** (**48 till 75**): These characters are best used for a font that only uses numbers
- **All** (**32 till 255**): These characters are used for a font that not only uses all of the characters in **Normal** but also extra ones, such as the copyright and registered trademark symbols
- **Letters** (**65 till 122**): These characters are used for a font that only includes letters and a few symbols, such as slashes
- **From Code**. This setting will examine and search for strings in all of

the game's code, creating several font ranges and using only those that are needed
- **From File**: This setting will bring up a file menu which will allow the user to select a text file that is searched, creating ranges only with characters found in the file

It is important to use an efficient font range because by eliminating unnecessary characters, less texture memory will be required, which will leave more room for other textures. For example, if a large font is desired, but this font is only going to be used to display a seven-letter word, then including only those seven letters is much more efficient than including the entire set of letters. On the other hand, using many characters can help prevent issues that may occur when you try to localize a game into different languages.

If the **–** button is pressed, the selected font range will be deleted, while the **Clear all** button will delete all of the font ranges.

> ## Note
>
> The `TextureGroup` dropdown menu is for a more advanced feature involving the placement of the letters in different textures to improve efficiency. More information about this dropdown can be found in GameMaker: Studio by navigating to **Help** | **Contents...**.

Creating the fonts

Now that the basics of font resource creation have been explained, the two fonts needed for the main menu—`fnt_main` and `fnt_title`—can be created. These two font resources will have identical settings except for their sizes. Their shared settings are as follows:

- **Font** should be set as **Arial**
- In the **Style** pane, **Anti-aliasing** should be set to **3**
- The option **Bold** should be checked
- **Font Range** should be set between **32** to **127**

The `fnt_main` font will use a size of `12`, while `fnt_title` will use a size of `36`.

Scripting obj_main_menu

As mentioned earlier, there are six events utilized by `obj_main_menu`. Many of these scripts appear to be rather lengthy; the reason for this is that when using the **Draw** events, just as when we create particle effects, there are many parameters that should be defined in advance.

Before creating these scripts, the contents of the main menu should be explained. We have this menu so that the player can adjust the following four main parameters of the puzzle game:

- Number of columns
- Number of rows
- Number of pieces
- The length of time played

In `obj_main_menu`, the adjustment of these settings will be done using four pairs of buttons, one button to decrease the value, and another to increase the value. A ninth button will be created that, when clicked on, will start the puzzle game with the defined parameters. All of these buttons will be rendered using `draw_button`. The main menu will also show the title of the game, which will be something short, silly, and quirky. The heading and value of each parameter will also be displayed.

The Create event

Because there are so many different elements being drawn in the **Draw** event, storing these elements is important. The instantiation of these variables will be done with three scripts in the **Create** event for the sake of organization. The first script executed will be as follows:

```
/// Initializes various instanced and global variables
for the main menu.

// The number of columns the game will use, ranging
from 6 to 20.
global.columns = 10;
```

```
column_min = 6;
column_max = 20;

// The number of rows the game will use, ranging from
6 to 12
global.rows = 7;
row_min = 6;
row_max = 12;

// The number of piece types that can be used in the
game ranging from 2 to 5.
global.piece_range = 3;
piece_min = 2;
piece_max = 5;

// The amount of time the player has to get the best
score possible in minutes.
global.start_time = 3;
start_time_min = 1;
start_time_max = 5;
```

The previous code defines global definitions of the number of columns and rows, the range of pieces, and the starting time for the game. The minimum and maximum values for these parameters are also defined.

The next script will define the positioning, the string, and other parameters of the text that will be drawn onscreen:

```
/// Initializes draw event information.

// Placement and text for the title.
title_info[0] = room_width  0.5;
title_info[1] = room_height  0.1;
title_info[2] = "PUZZLERS!";

// Placement and text for the main heading.
heading_info[0] = title_info[0];
heading_info[1] = room_height  0.3;
heading_info[2] = "Game Options";

// Column Info.
column_info[0] = title_info[0];
column_info[1] = room_height  0.4;
column_info[2] = "Columns:  " +
string(global.columns);
```

```
// Row Info.
row_info[0] = title_info[0];
row_info[1] = room_height  0.45;
row_info[2] = "Rows:  " + string(global.rows);

// Start time draw info.
time_info[0] = title_info[0];
time_info[1] = room_height  0.5;
time_info[2] = "Time:  " + string(global.start_time) +
" Minutes";

// Placement and text for the piece info.
piece_info[0] = title_info[0];
piece_info[1] = room_height  0.55;
piece_info[2] = "Pieces";


// Parameters for displaying the pieces that will be
in the level.
piece_width = sprite_get_width(spr_pieces);
piece_width_mid = global.piece_range  piece_width
0.5;
piece_pos_x_mid = title_info[0] - piece_width_mid;
piece_pos_y = room_height  0.7;
```

In the previous code, most of the information is stored in arrays with three components: an *x* coordinate, a *y* coordinate, and a string. Additionally, when defining the coordinates, `room_width` and `room_height` are multiplied by percentages. This is a useful approach when defining the placement of objects in case you want to change the size of the room.

The final set of instanced variables deals with the placement of piece sprites. Instead of just displaying the number of pieces that will be used in the game with text, the sprites themselves will be shown.

The width of the piece sprite is used for spacing. Then, the total width is determined by calculating the product of `piece_width` and the number of pieces that will be used. This number is multiplied by `0.5` since we really just want to find the center. This is because subtracting half of the displayed pieces' total width from the center of the room will help keep the pieces centered when they are displayed in the menu.

The following code is the final part of the **Create** event, which will handle placement for the buttons:

```
/// Initializes the draw button info.

// String displayed on GUI buttons to increment a
value.
inc_string = "+";

// String displayed on GUI buttons to decrement a
value.
dec_string = "-";

// Defines the half sizes of the small buttons.
var button_sm_width = 32;
var button_sm_height = 12;

// Defines the half sizes of the large buttons.
var button_lg_width = 128;
var button_lg_height = 32;

// Defines the middle x and y coordinates of the 8
buttons.
button_pos_mid[0] = room_width  0.35;
button_pos_mid[1] = room_width  0.65;

// Uses a for loop to define the button information.
for (var i = 0; i < 8; i++)
{
    // mod is used to determine if the button is on
the left (0) or on the right (1).
    var i_mod = i mod 2;
    button_x[i, 0] = button_pos_mid[i_mod];

    // The width for the small button are subtracted
and added from the x coordinate of the center.
    button_x[i, 1] = button_x[i, 0] - button_sm_width;
    button_x[i, 2] = button_x[i, 0] + button_sm_width;

    // div is used to move each pair of buttons onto a
new row.
    button_y[i, 0] = room_height  (0.4 + 0.05  (i div
2));

    // The height for the small button are subtracted
and added from the y coordinate of the center
```

```
    button_y[i, 1] = button_y[i, 0] -
button_sm_height;
    button_y[i, 2] = button_y[i, 0] +
button_sm_height;

    // A variable used to determine if the button is
currently down or not.
    button_down[i] = false;

    // If on the left, the text is set to the minus
sign, the plus sign is used.
    if (i_mod == 0)
    {
        button_text[i] = dec_string;
    }
    else
    {
        button_text[i] = inc_string;
    }
}

// Defines parameters for the play button using the
large button width and height.
button_x[8, 0] = room_width  0.5;
button_x[8, 1] = button_x[8, 0] - button_lg_width;
button_x[8, 2] = button_x[8, 0] + button_lg_width;

button_y[8, 0] = room_height  0.85;
button_y[8, 1] = button_y[8, 0] - button_lg_height;
button_y[8, 2] = button_y[8, 0] + button_lg_height;

button_down[8] = false;
button_text[8] = "Start Playing!";

// Used for keyboard input, highlighting which row of
buttons is being pressed.
current_btn_row = noone;
```

This is the final portion of code within the **Create** event. Many variables for the placement of the main menu's buttons are defined here. The `for` statement is used to initialize these variables without having to type them all out.

Now, this seems like a lot of code, but all of these variables are predefined for efficiency. Instead of having to determine the placement of

the buttons or initializing new strings every time the **Draw** event is called, these buttons and strings are predefined so that they just have to be called. The numbers used in the previous block of code, particularly those multiplied by `room_width` or `room_height`, may seem arbitrary, but they were tested and adjusted multiple times with the final layout in mind.

Utilizing div and mod

In the previous section, two keywords are introduced: `div` and `mod` (integer division and modulo). The `div` function returns a whole number with no remainders. For example, 1 divided by 2 is 0.5. The whole number remaining is 0, so `1 div 2` would be `0`. In the previous block of code, `div` is used in the following formula:

```
button_y[i, 0] = room_height  (0.4 + 0.05  (i div 2))
```

In this example, `div` is used to keep pairs of buttons in the same row. The `0 div 2` and `1 div 2` expressions will both return `0`, so `button_y[0, 0]` will be equivalent to `button_y[1, 0]`, and so on as `i` is incremented. Meanwhile, `mod` returns only the remainder of a division operation. For example, when 1 is divided by 2, the remainder is 1, so `1 mod 2` will return `1`; on the other hand, when 8 is divided by 2, the result is 4, but there's no remainder, so `8 mod 2` will return `0`. The `mod` function is used when establishing button positions with the following code:

```
// mod is used to determine if the button is on the
left (0) or on the right (1).
    var i_mod = i mod 2;
    button_x[i, 0] = button_pos_mid[i_mod];
```

In the previous code, `i mod 2` will return either `0` or `1`. In this case, `mod` determines whether a button should use index `0` or `1` of the array `button_pos_mid`, essentially determining which buttons go to the left and which ones go to the right of the screen.

The `div` and `mod` functions are two very useful functions, especially when incrementing and trying to organize values in a specific manner, such as the button layout for `obj_main_menu`.

Drawing the menu

Now that all of the necessary parameters have been defined, the scripting of the **Draw** event can begin.

> ## Note
>
> Because the main menu is the only element being drawn on the screen and the view has not been changed, it is irrelevant whether or not a **Draw** event or a **Draw GUI** event is used.

The following code will first draw all of the main text defined in the second **Create** event script and then all of the buttons defined in the third script:

```
/// Draws all of the GUI for the main menu.

// Sets the font and alignment for the title.
draw_set_font(fnt_title);
draw_set_halign(fa_center);
draw_set_valign(fa_middle);

// Draws the title.
draw_text(title_info[0], title_info[1],
title_info[2]);

// Sets the font for the main text.
draw_set_font(fnt_main);

// Draws the heading and other parameter texts.
draw_text(heading_info[0], heading_info[1],
heading_info[2]);
draw_text(column_info[0], column_info[1],
column_info[2]);
draw_text(row_info[0], row_info[1], row_info[2]);
draw_text(time_info[0], time_info[1], time_info[2]);
draw_text(piece_info[0], piece_info[1],
piece_info[2]);
```

In the previous code, all the text is drawn. The font and alignments are

first set before drawing the title, using info from the array—`title_info`—created earlier. Then, after the font is changed, all of the different strings are drawn onscreen. Note that the alignments are not updated as we want to keep them the same when drawing all of this text. The following code draws the sprites that will show up in the level:

```
// Draws the sprites that will show up in the level.
for (var i = 0; i <= global.piece_range; i++)
{
    draw_sprite(spr_pieces, i, piece_pos_x_mid +
piece_width * i, piece_pos_y);
}

// Draws the buttons.
for (i = 0; i < 9; i++)
{
    // Gets which row the button is in.
    var button_row = i div 2;

    // If the button is in the current row, it is
colored aqua; otherwise, it is light gray.
    if (current_btn_row == button_row)
    {
        draw_set_color(c_aqua);
    }
    else
    {
        draw_set_color(c_ltgray);
    }

    // The button is drawn.
    draw_button(button_x[i, 1], button_y[i, 1],
button_x[i, 2], button_y[i, 2], !button_down[i]);

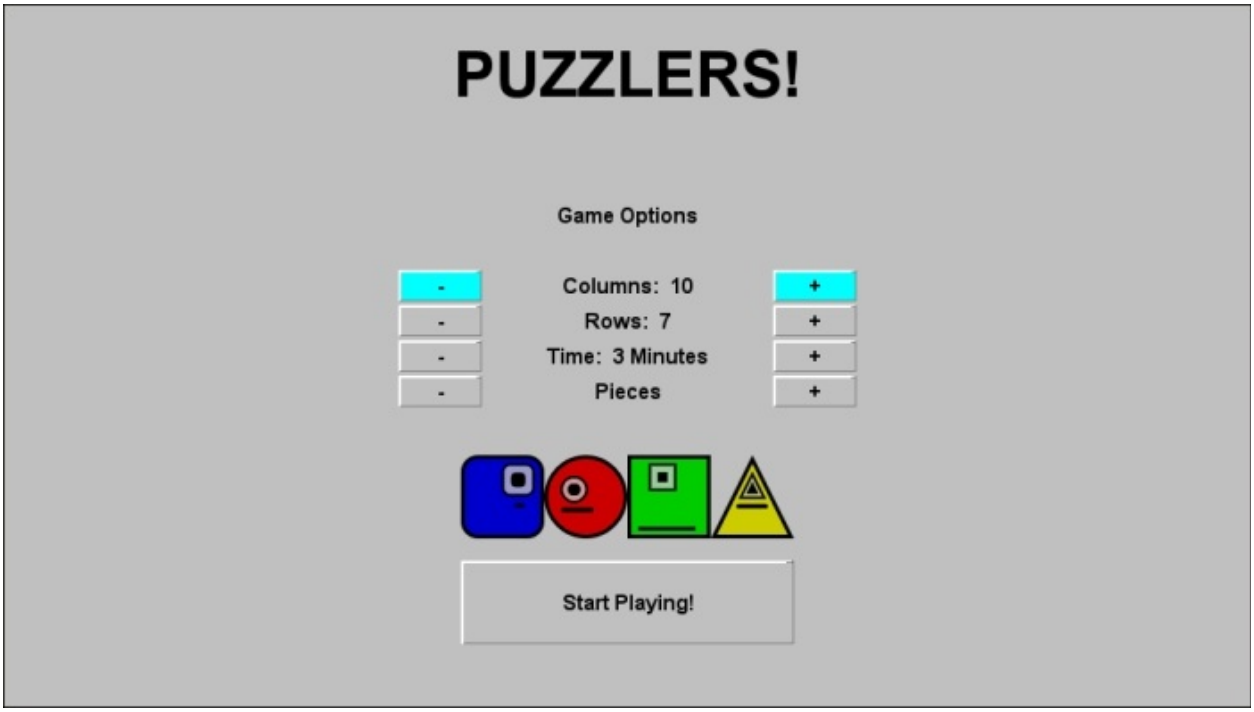
    // The color is set to black for the text.
    draw_set_color(c_black);

    // Draws the centered text at the mid point of the
button.
    draw_text(button_x[i, 0], button_y[i, 0],
button_text[i]);
}
```

After drawing the text, this second block of code first draws the pieces

that will be used within the level by running a `for` statement. Then, the nine buttons are drawn. The `div` function is used once again to determine which row the user is currently highlighting. The button is then drawn with `draw_button`, using `!button_down` to check whether or not the button is in the up state and the button's text is drawn over it.

If set up properly, the main menu should now look something like that shown in the following screenshot:



This is a good start, but now, interaction needs to be added, which will be done using the mouse and keyboard events discussed earlier.

The Global Left Button event

Now that the main menu's buttons are visible, player interaction with them must be scripted. The **Global Left Button** mouse event will trigger every frame for which the left mouse button is held down. The global event must be used because `obj_main_menu` has no information about the collision mask or bounds. The drawn buttons will be tested using the following code:

```
// Resets any highlighting performed by the keyboard.
current_btn_row = -1;

// Test to see which button the mouse is in range of.
for (var i = 0; i < 9; i++)
{
    // Determines if mouse_x and mouse_y are within
the x and y coordinates of the button.
    button_down[i] = (mouse_x > button_x[i, 1] &&
mouse_x < button_x[i, 2] && mouse_y > button_y[i, 1]
&& mouse_y < button_y[i, 2]);

    // If the button is down, that row is set to
highlight.
    if (button_down[i])
    {
        current_btn_row = i div 2;
    }
}
```

In the previous code, `current_btn_row` is set to `-1`, meaning that no row should currently be active. Then, the nine individual buttons are iterated through the use of a `for` statement. Within each iteration, the values of `button_down` are set by a compound Boolean.

The test makes sure that `mouse_x` and `mouse_y` are within the predefined bounds of the various buttons. In `button_x`, the first dimension of the array is the button index. The second dimension uses three parts to represent the button, as shown in the following list:

- `0` represents the center button
- `1` represents the left or top button
- `2` represents the right or bottom button

If the corresponding mouse points fall within the previously mentioned ranges, we can assume that a respective button is currently being pressed down. Finally, if the button is held down, `current_btn_row` is set to `i div 2`, so both buttons are highlighted when either button in this row is within range.

The press <any key> event

Before moving on to the other mouse events, the **press <any key>** event will be implemented since it is aimed toward achieving the same goal as the **Global Left Button** event. The script is as follows:

```
/// Changes the state of the drawn buttons with the
keyboard.

// Sets the current row to at least 0.
current_btn_row = max(current_btn_row, 0);

// Sets different buttons based on what was pushed.
switch (keyboard_key)
{
    // If up is pushed.
    case vk_up:
        // Moves to the previous button row, and if
less than 0 the value loops to its maximum value.
        current_btn_row--;
        if (current_btn_row < 0)
        {
            current_btn_row = 4;
        }
        break;
    case vk_down:
        // Moves to the next row, and if greater than
the maximum value, loops around to 0.
        current_btn_row++;
        if (current_btn_row > 4)
        {
            current_btn_row = 0;
        }
        break;
    case vk_left:
        // Sets the button on the left to its down
state.
        if (current_btn_row < 4)
        {
            button_down[current_btn_row * 2] = true;
        }
        break;
    case vk_right:
        // Sets the button on the right to its down
state unless the play button is highlighted.
        if (current_btn_row < 4)
```

```
            {
                button_down[current_btn_row  2 + 1] =
true;
            }
            break;
        case vk_space:
        case vk_enter:
            // Puts the play button in its highlight
state.
            if (current_btn_row == 4)
            {
                button_down[current_btn_row  2] = true;
            }
            break;
}
```

In this state, the buttons are adjusted by different keyboard events. In the `switch` statement, the keyboard in which the key is currently pressed is tested with several values. `vk_up` and `vk_down` will loop through the different rows of buttons. Meanwhile, `vk_left` and `vk_right` will highlight the buttons that will increase and decrease the different main menu objects. The correct button index is obtained by multiplying the current row by `2` if the row is on the left and adding 1 to that result if the row is on the right, which essentially reverses the `div` and `mod` executions performed to organize the buttons. This is not done, however, if the start button is highlighted since there is only one of its kind. Instead, `vk_space` or `vk_enter` will trigger the down state for the player button.

The Global Left Release event

Now that the buttons can be set to the down state by either the keyboard or mouse, the buttons have to be set to the release state using both means. Also, these buttons must trigger a reaction so that the associated values can actually be updated. The following code is used to release the left mouse button:

```
// Checks each button.  If the button was down, the
menu press script       is executed.
for (var i = 0; i < 9; i++)
{
```

```
        if (button_down[i] && ((i div 2) ==
    current_btn_row))
        {
            scr_on_menu_button_pressed(i);
        }

        // Sets all buttons to being down to false.
        button_down[i] = false;
    }
```

In the previous code, the nine buttons are iterated through and any button that is held down will call `scr_on_menu_button_pressed` using the current index as the only argument. Also, all the values in the array `button_down` are set to `false`, so the buttons appear to be released.

The release <any key> event

Now, when a key is released, the reaction seen on the screen is actually identical to that when the mouse is released. Instead of copying and pasting the script from the **Global Left Release** event, a new script resource could be created; however, the following code will be utilized to showcase the unique built-in function `event_perform`:

```
// Triggers the global left release event.
event_perform(ev_mouse, ev_global_left_release);
```

event_perform

The built-in function `event_perform` is a special function that allows other events to be called by the object. Its arguments consist of different built-in constants that represent each of the different events that can be created in GameMaker: Studio. Some event types such as `ev_create` and `ev_destroy`, which represent the **Create** and **Destroy** events respectively, do not have any event numbers and `0` can be assigned to `argument1`.

Note

The full list of these constants which reflect the events that can be

created in the **Object Properties** panel and can be found in GameMaker: Studio by navigating to **Help** | **Contents...**.

scr_on_menu_button_pressed

The final step in giving final touches to the main menu is coding the script executed by the **Global Left Release** and **release <any key>** events. Using the index of the triggering button, a `switch` statement is used to adjust all of the different parameters for the game, as shown in the following code:

```
// Triggers the appropriate action by using the index
of the button pressed.

switch (argument0)
{
    // Decreases column.
    case 0:
        global.columns = max(global.columns - 1,
column_min);
        column_info[2] = "Columns:  " +
string(global.columns);
        break;
    // Increases column.
    case 1:
        global.columns = min(global.columns + 1,
column_max);
        column_info[2] = "Columns:  " +
string(global.columns);
        break;
    // Decreases row.
    case 2:
        global.rows = max(global.rows - 1, row_min);
        row_info[2] = "Rows:  " + string(global.rows);
        break;
    // Increases row.
    case 3:
        global.rows = min(global.rows + 1, row_max);
        row_info[2] = "Rows:  " + string(global.rows);
        break;
    // Decreases start time.
    case 4:
```

```
        global.start_time = max(global.start_time - 1,
start_time_min);

        time_info[2] = "Time:  " +
string(global.start_time);

        // Sets the text to use proper grammar if only
one minute remains.
        if (global.start_time == 1)
        {
            time_info[2] += " Minute";
        }
        else
        {
            time_info[2] += " Minutes";
        }
        break;
    // Increases start time.
    case 5:
        global.start_time = min(global.start_time + 1,
start_time_max);
        time_info[2] = "Time:  " +
string(global.start_time) + " Minutes";
        break;
    // Decreases piece range.
    case 6:
        global.piece_range = max(global.piece_range -
1, piece_min);
        piece_width_mid = global.piece_range
piece_width  0.5;
        piece_pos_x_mid = title_info[0] -
piece_width_mid;
        break;
    // Increases piece range.
    case 7:
        global.piece_range = min(global.piece_range +
1, piece_max);
        piece_width_mid = global.piece_range
piece_width  0.5;
        piece_pos_x_mid = title_info[0] -
piece_width_mid;
        break;
    // Goes to the game room.
    case 8:
        room_goto(rm_gameplay);
        break;
}
```

In the previous code, whenever a value is decreased, `max` is used to make sure it doesn't go below the predefined minimum value; meanwhile, `min` is used to make sure the value doesn't exceed the predefined maximum value. Also, the appropriate text is defined similarly to the way it was in the **Create** event, except for `time`, which will check if the value for `global.start_time` is `1`, using the text `Minute` instead of `Minutes`. Though the game would still run fine with this spelling mistake, this small detail makes the game appear more competent. Finally, if the play button is highlighted (`case 8`) the game goes to the main game room `rm_gameplay` using the built-in function `room_goto`.

If all of these scripts have been implemented properly, the player should now be able to adjust various parameters on the main menu screen, staying within a certain range. They'll also be able to gain access to the game when they're ready to play. If they do, however, they'll quickly notice that their changes haven't done anything. This is because the previously mentioned scripts have not been implemented in the game, which will be done next.

Changing obj_grid_manager

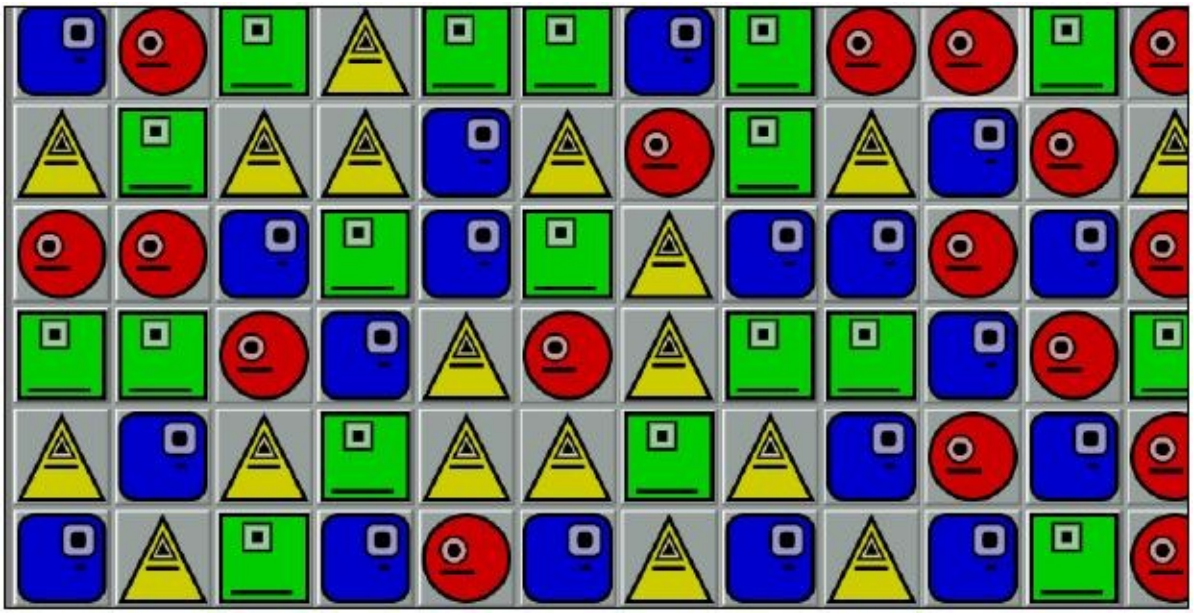
Now that parameters can be changed with `obj_main_menu`, those changes must be integrated with `obj_grid_manager`. One change is within its **Create** event. Where the instanced variables `columns`, `rows`, and `piece_range` are defined, a slight change is required, as shown in the following code:

```
// Defines the number of columns in the grid.
columns = global.columns;

// Defines the number of rows in the grid.
rows = global.rows;

// The number of pieces that will be present on the
board.  Should be greater than 1.
piece_range = global.piece_range;
```

These three values are now set to the global values set by `obj_main_menu`. If these values are adjusted, the changes will be immediately reflected in the game; however, something is off. Because the pieces are placed at the `x` and `y` coordinates of `obj_grid_manager`, the game is off center, and sometimes pieces are moved off the screen, as demonstrated in the following screenshot where 20 columns and 6 rows are used:



To remedy the problem of the game going off center and sometimes pieces being moved off the screen, additional changes will be made to the previously changed **Create** event script. The following is the change to the **Create** event script, which begins after the instanced variables `x_spacing` and `y_spacing` are defined:

```
// Uses the original placement to determine the
desired distance from all sides.
var x_margin = x;
var y_margin = y;

// Determines the width and height of the board.
var width = x_spacing * columns;
var height = y_spacing * rows;

// Gets the width of the room using the predefined
margins.
var room_width_margin = room_width - x_margin * 2;
var room_height_margin = room_height - y_margin * 2;

// If either size exceeds the size of the room, the
scale must be adjusted.
if (width > room_width_margin || height >
room_height_margin)
{
    image_xscale = room_width_margin / width;
    image_yscale = room_height_margin / height;     //
```

```
Sets the scale to the smallest value in equal
proportions.
    image_xscale = min(image_xscale, image_yscale);
    image_yscale = min(image_xscale, image_yscale);
}

// Multiplies the spacing by the updated scale.
x_spacing *= image_xscale;
y_spacing *= image_yscale;

// The board is moved to fit the new sizing;
width *= image_xscale;
height *= image_yscale;

// The board is moved so pieces are placed at the
right starting point.
x = x_margin + (room_width_margin - width + x_spacing)
* 0.5;
y = y_margin + (room_height_margin - height +
y_spacing) * 0.5;

// Define variables for loops.
var i, j;

// Loop through the columns and rows, creating the
grid and puzzle pieces, now using the scale.
for (i = 0; i < columns; i++)
{
    for (j = 0; j < rows; j++)
    {
        array_grid[i, j] = scr_create_in_grid(x, y,
x_spacing, y_spacing, 1, obj_grid_block, i, j,
image_xscale);
        array_pieces[i, j] = scr_create_in_grid(x, y,
x_spacing, y_spacing, 0, obj_puzzle_piece, i, j,
image_xscale);
        array_pieces[i, j].image_index =
irandom(piece_range);
    }
}
```

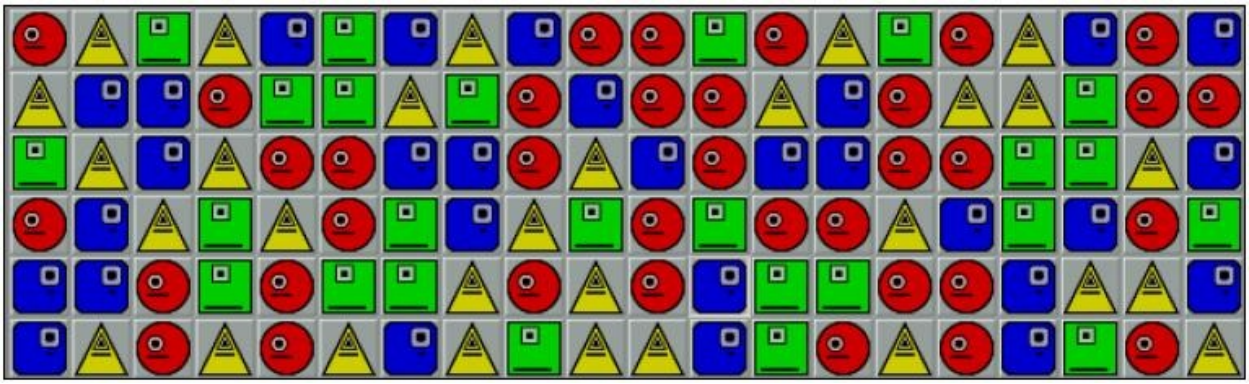
This script determines how much the board needs to be scaled based on the new size that the board will exhibit with different values of rows and columns. The `x` and `y` coordinates are also shifted so that the board is always centered, even if scaling does not occur. If the new width or

height exceeds the size of the room minus the respective margin on all of the sides, the scale of `obj_grid_manager` is adjusted and used as an additional argument within `scr_create_in_grid`.

A change to the `scr_create_in_grid` script from Chapter 2, *Random Organization – Creating a Puzzle Game*, will also be required. The change to this script resource is simple. Essentially, a new argument is used that will change the scale of the newly created pieces. Since `image_xscale` and `image_yscale` have equivalent values, only one argument needs to be passed. The following code should be added to `scr_create_in_grid` after the variable `new_instance` is assigned:

```
// Adjusts the object's scale.
new_instance.image_xscale = argument8;
new_instance.image_yscale = argument8;
```

If these changes are made properly, a board with 20 columns and 6 rows should now look something like that shown in the following screenshot:



The pieces are smaller, but their functionality remains the same and the game should be playable just as it was before.

Integrating score and time

Being able to edit the parameters of each play session of the puzzle game is nice, but the game is still lacking cumulative rewards for creating matching sets as well as building tension. In the next section, elements will be implemented and integrated to add these important features.

Scoring and combos

Scoring can accomplish a variety of goals in any game. In this puzzle game, it will give the player a sense of progress and achievement, and they can test themselves to see how high a score they can achieve each session.

By this design, players will earn 100 points for every piece they break in a match. In addition, each consecutive series of breaks increases the score further; this will be called a **combo**. For example, if a player matches three pieces and after falling into place, the pieces from above the matched pieces create a new match, the player will earn 200 points for each piece broken.

The Create event of obj_grid_manager

To begin accounting for scoring and combos, some variables must be initiated in the **Create** event of `obj_grid_manager`, as shown in the following code:

```
// Resets the current score to 0.
score = 0;

// Combo counter used to exponentially increase the
score for consecutive matches.
combo_count = 0;

// Defines information for the score.
score_info[0] = room_width  0.03;
score_info[1] = room_height  0.3;
```

```
score_info[2] = "Score:#0000000000";
```

Now, `score` is a special built-in global variable included within GameMaker: Studio and GML because scoring is such a common concept in games. It is set to `0` in the **Create** event so that each play through restarts it. The instanced variable `combo_count` is also set to `0` for the same reason. Then, similar to drawn elements in the main menu, an array is initialized storing the `x` and `y` coordinates and the string.

> ## Note
>
> The character `#` is used to create a line break in strings.

Earning points in scr_reorganize_board

Now that the required variables have been initialized, the score and the appropriate string value can now be increased. This will be done in the script `scr_reorganize_board`. The additions to this script will be made after the additions are made to the pair of `for` statements that tests to see which pieces make a match:

```
// Increments the combo count.
combo_count++;

// Loops through the newly created array.
for (i = 0; i < array_length; i++)
{
    // Assign the local variable since it will be
used.
    piece = array_broken_piece[i];

    // Sets matched to false.
    piece.matched = false;

    // Move the piece off-screen.
    piece.y = -100;

    // The piece is continually swapped with pieces
above it until it reaches the top.
```

```
    while (piece.row != 0)
    {
        // Continually move the piece up so the pieces
fall in order.
        piece.y -= 100;

        // The fourth argument is -1 as it is known
which direction the swap will take place.
        scr_swap_pieces(piece.col, piece.row, 0, -1,
true, 1);
    }

    // Set first alarm to 1 so pieces at row 0 will
still fall into place.
    piece.alarm[1] = 1;

    // 100 points are added for each broken piece.
    score += 100 * combo_count;

    // Sets the image_index to give the illusion of a
new piece being created.
    piece.image_index = irandom(piece_range);
}

// Adjusts the score and piece count texts.
var score_string = string(score);

// Add zeros in front of the score string until it is
10 characters long.
while (string_length(score_string) < 10)
{
    score_string = "0" + score_string;
}
score_info[2] = "Score:#" + score_string;

// Checks the board for new matches.
if (!scr_check_board(true))
{
    combo_count = 0;
    shifting_pieces = false;
}
```

In the previous code block, the additions to the script have been highlighted. The first one increments `combo_count`. This will ensure that each consecutive break will increase the combo counter.

Then, within the `for` statement, the score is increased by multiples of 100 and `combo_count`. It's important to note that `combo_count` should be incremented before this `for` statement as not doing so would mean the player wouldn't earn any points for their first match.

After setting the score, the string used to display it has to be updated. First, the value is converted to a string using the built-in function `string`. The next portion is to polish the game; sometimes all the zeroes in a score are shown to hint at how high a player's score can go or to give them something to reach for. For this game, the score string will be ten digits long. To do this, a `while` statement is used that tests the string's length, or the number of characters within it. This can be obtained using the function `string_length`. If `string_length` is less than `10`, `0` is added in front of the current string. The value of the second index `score_info` is set as a value that is the combination of this 10-digit string and the second index that was used in the **Create** event.

Finally, if no more matches are found after executing `scr_check_board`, the `combo_count` object is set back to `0`.

Drawing UI in obj_grid_manager

Now that the score is being incremented, it needs to be displayed. To do this, a **Draw** event must be utilized. Since `obj_grid_manager` is managing so many other aspects of the game, it makes sense to add a **Draw** event to it. Again, the choice between **Draw** and **Draw GUI** is irrelevant since the game view isn't changing. The following code will display the score text, setting the font color to `c_black`, and aligning it to the left side of the screen and using the font `fnt_main`:

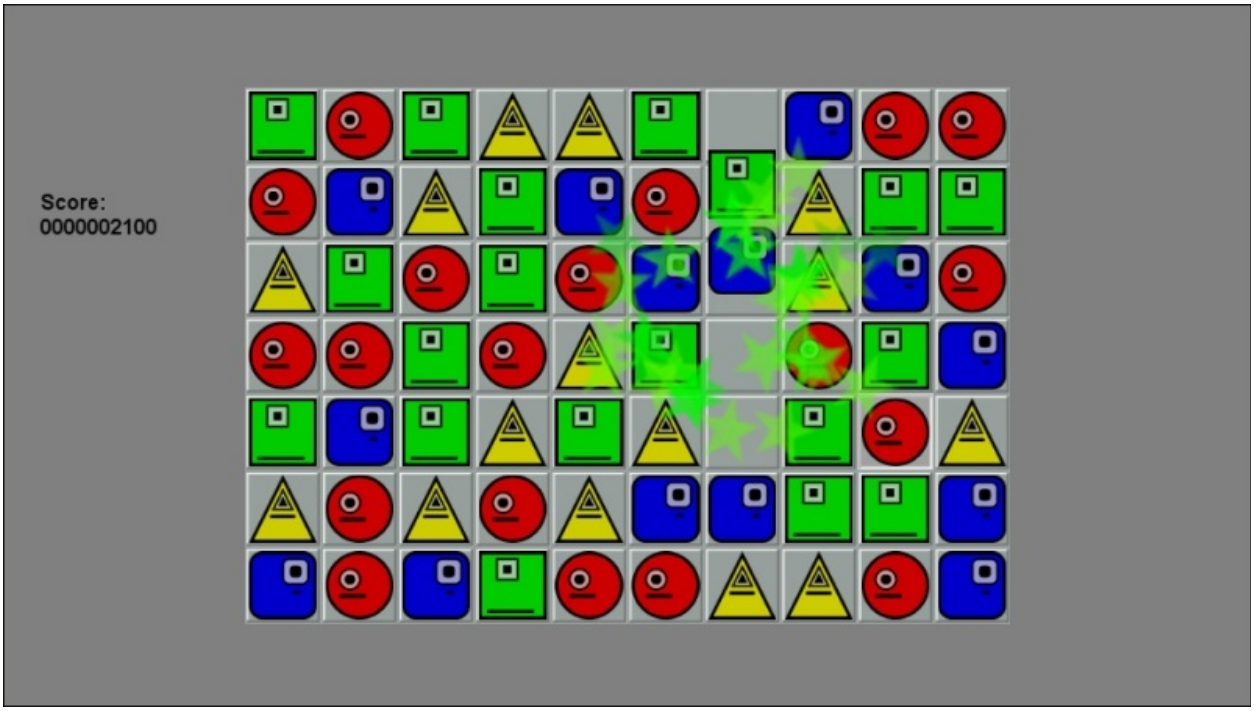
```
/// Draw's the board's UI.

/// Sets the font and alignment
draw_set_font(fnt_main);
draw_set_halign(fa_left);
draw_set_valign(fa_middle);
draw_set_color(c_black);

// Draws the score on the left of the screen.
```

```
draw_text(score_info[0], score_info[1],
score_info[2]);
```

Upon the completion of this code, the player should now be able to play the game and be rewarded points as shown in the following screenshot:



Timing and game over

Now that the score has been integrated into the puzzle game, timing can be as well. Time is one of the many types of pressure that can make a game more exciting, add tension, and alleviate dullness. In this section, alarms will be utilized to track time. The timer will then be displayed across the bottom of the game screen. A "game over" state will also be triggered to serve as feedback and display the player's final score more boldly before returning to the main menu.

Adding new variables to the Create event of obj_grid_manager

This chapter is full of the initialization of a lot of variables, and to begin

integrating time, more variables will be introduced. The following code can be written at the end of the **Create** event scripts where the scoring variables are initialized:

```
// The total time in the level.
alarm[1] = global.start_time * room_speed  60;

// Information for the timer, which will be displayed
as a rectangle that transitions through three colors.
timer_info[0] = room_width  0.1;
timer_info[1] = room_height  0.92;
timer_info[2] = room_width  0.9;
timer_info[3] = room_height  0.96;

timer_percentage = 100.0  alarm[1];

timer_width = timer_info[2];

/ Game over variables
game_over_info[0] = room_width  0.5;
game_over_info[1] = room_height * 0.5;
game_over_info[2] = "TIME OVER!##FINAL SCORE#0";
```

Firstly, the value of `alarm[1]` is set to the product of `room_speed`—`60`—and `global.start_time`. Setting an alarm to the built-in variable, `room_speed`, is equivalent to setting the alarm to `1` second. Since the main menu designates the values in minutes, multiplying the room speed by the product of `global.start_time` and `60` will set the alarm to the appropriate amount in seconds.

The four values stored in `timer_info` are the two `x` and two `y` coordinates used for drawing the timer across the bottom of the screen in the **Draw** event. The function, `draw_healthbar` will be utilized to do this; though the time and health are different things, the timer acts as though it is slowly draining health. Then, a percent value is derived and assigned to `timer_percentage`. Since `draw_healthbar` requires a value that lies between `0` and `100` to determine how full the bar is, dividing `100` by the value of `alarm[1]` will get this value.

Finally, information for the text that will be shown when the timer runs out is set up in an array called `game_over_info`. This array will show the text

and the final score the player has achieved all within one string.

Using Alarm 1 and Alarm 2 in obj_grid_manager

Now that new variables have been initialized, two **Alarm** events will be created. The first alarm at the index `1` will be triggered when the time set previously reaches `0`. The alarm at the index `2` will be used to add a slight delay in the game, so the player can see their final score before being sent back to `rm_main_menu`.

Alarm 1

The **Alarm 1** event will handle ending the game in a few different ways. It is also important to make sure **Alarm 1** is set up with a script of some kind because the **Alarm** events with no scripts or actions attached will not run, even if values are assigned to them. The following is the script for **Alarm 1**:

```
/// Ends the game.

// Sets this value to true so pieces cannot be
shifted.
shifting_pieces = true;

// Iterates through each piece.
for (var i = 0; i < columns; i++)
{
    for (var j = 0; j < rows; j++)
    {
        // Hides the grid.
        array_grid[i, j].visible = false;

        // Sets a random speed and gravity to give the
illusion of popping off the board.
        with (array_pieces[i, j])
        {
            hspeed = random_range(-10, 10);
            vspeed = random_range(-5, -25);
            gravity = 1;
```

```
            // Both alarms are set to 0 to prevent any
movement that might have been occurring when this
alarm was triggered.
            alarm[0] = 0;
            alarm[1] = 0;
        }
    }
}

// Sets the string for the game over value.
game_over_info[2] = "TIME OVER!##FINAL SCORE#" +
string(score);

// fades out the music.
audio_music_gain(0, 2500);

// Sets the alarm to five seconds.
alarm[2] = room_speed * 5;
```

In the previous block of code, `shifting_pieces` is set to `true`; this is done to prevent the player from attempting any more swaps. Then, every grid block is hidden by setting `visible` to `false`, and every piece is iterated in the game and assigned a random speed both horizontally and vertically. Their `gravity` value is also set to `1`, so they accelerate downwards. This will cause all of the pieces on the board to fly off, acting as feedback and informing the player the session has ended. Finally, the two alarms that handle the translation of pieces are set to `0` to prevent them from interfering while the pieces fall off the screen.

After iterating through all the pieces, the string showing that the game is over is set, informing the player that the round has ended and showing them their final score. Also, the music is faded out. Again, the second argument of `audio_music_gain` uses milliseconds to determine the amount of time, so a value of 2,500 should be about two and a half seconds.

Finally, the second alarm is set to the product of `5` and `room_speed`, about 5 seconds.

Alarm 2

The code within the second alarm is very simple and will handle two things: stopping the music and returning to the main menu room. The music is stopped to make sure that when the game returns to `rm_main_menu`, that room is in a near-identical state to what it was when the game was started. In this case, music wasn't playing, so the music that should have faded out to silence by now is stopped. The following is the script for **Alarm 2**:

```
/// Goes to the main menu room.

// Makes sure to stop the music.
audio_stop_music();

// Goes to the main menu.
room_goto(rm_main_menu);
```

Drawing the timer

Now that the required variables and the appropriate alarms have been set up, the timer and the text showing that the game is over can be drawn in the **Draw** event of `obj_grid_manager` by adding the following code to the end of its script:

```
// Determines the percentage of the timer while making
sure it doesn't fall below 0.
var percent = max(0, alarm[1] * timer_percentage);

// Draws the custom timer and sets the color.
draw_healthbar(timer_info[0], timer_info[1],
timer_info[2], timer_info[3], percent, c_black, c_red,
c_green, 0, true, true);

// If the percent is less than or equal to 0, meaning
the round is over, game over text is written.
if (percent == 0)
{
    draw_set_font(fnt_title);
    draw_set_valign(fa_middle);
    draw_set_halign(fa_center);
    draw_text(game_over_info[0],
game_over_info[1],game_over_info[2]);
}
```

First, a local variable—`percent`—is defined, which takes the value of `alarm[1]` and multiplies it by `timer_percentage`, which should return a value between `0` and `100`; `max` is used to make sure the value doesn't fall below `0`.

After determining this value, it is used within the built-in function `draw_healthbar` as well as the previously defined four components of `timer_info`.

Finally, when `percent` is equal to `0`, the text showing that the game is over is rendered, its alignment and fonts set beforehand.

If all of the scripts prior to this script have been implemented properly, the player should be able to swap pieces and receive points for doing so. A timer should also exist on the bottom of the screen that slowly drains, and when it runs out, the pieces should fly off the board, displaying the player's final score, as shown in the following screenshot:



Summary

In this chapter, the powerful **Draw** and **Draw GUI** events were expanded upon and used to create an interactive main menu and display scores and a decreasing timer. All of this helped give the puzzle game a more complete feel, allowing the player to achieve rewards, feel pressure, and replay the game if they desire. Now, this puzzle game is by no means a finished product and was never meant to be. It was meant to serve as a starting point and to demonstrate different aspects of GameMaker Language. There are still tons of features that could be included, such as tutorials, leaderboards, achievements, and power ups, just to name a few. Also, to make this a finished product, it would require a lot more polish, a better title, and more importantly, something to make it stick out more from all the different types of match-three games that exist in the market today.

From Chapter 2, *Random Organization – Creating a Puzzle Game* until this current chapter, many different useful, built-in functions, variables, and syntax rules of GMLs were listed. This puzzle game, however, was also created to help make this process a little more entertaining and playful. The following list shows what we did in the previous chapters:

- Chapter 2, *Random Organization – Creating a Puzzle Game*, introduced setting up the game and organizing pieces randomly on the board
- Chapter 3, *So How Do I Play? – Adding Player Interaction*, utilized player input from both the mouse and keyboard
- Chapter 4, *Juicy Feedback – Aural and Visual Effects*, demonstrated the creation of feedback through particle and sound effects

Instead of continuing the puzzle game and attempting to make it more like a shippable product in the next chapter, a new game project will be created, introducing a more active genre: the platformer! The remainder of this book will cover features for this genre, such as game views, collision, and simple artificial intelligence. The next chapter, however, will have more humble beginnings, introducing the concept of a finite state machine and character movement.

Chapter 6. Finite State Machines – Starting the 2D Platformer

In the previous chapter, **Draw** events were used to create a menu as well as display score and timer information for the puzzle game. The production of new features for that game, however, will conclude in favor of starting a new game. In this chapter, and for the remainder of this text, a 2D platformer will be built!

The average 2D platformer deals with twitch-based gameplay that consists of a character usually running and jumping around in an environment. This environment is then filled with various hazards, such as spikes, pitfalls, and monsters. In this first chapter, the focus will be on setting the state of the character through the use of a finite state machine. By the end of this chapter, the following aspects of the platformer will be set up:

- A platformer character will be created
- This character will transition from the idle state to the walking or jumping state
- These state transitions will be made through keyboard input
- The character will also be moved through the use of built-in variables, such as friction and gravity

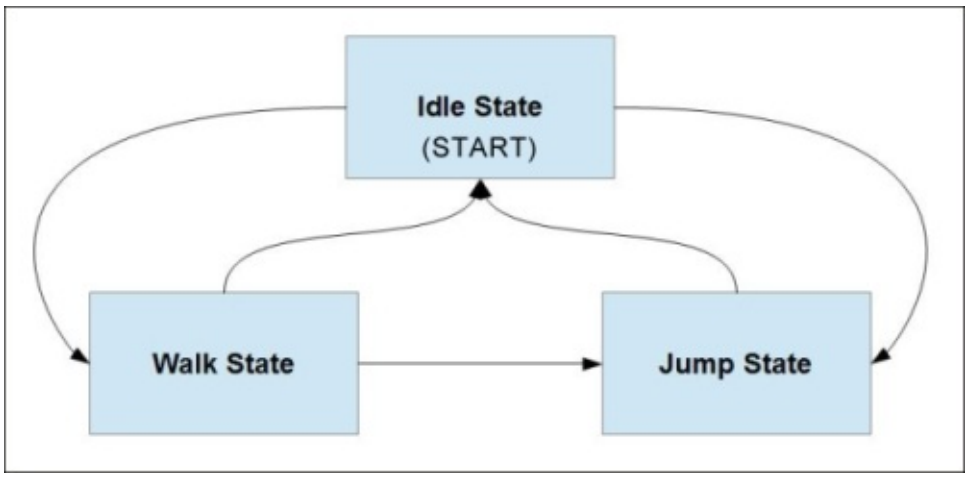
Introducing finite state machines

A **finite state machine** (**FSM**) is an abstract programming concept involving the management of an object, its different states, and the transition between those states. A **state** can be defined as a unique mode or condition that an object is in at a given time.

For example, a door can have various states: opened, closed, opening, and closing. What an FSM does is make sure the door is only in one state at any given time. A door can't be both open and closed at the same time, right? In addition, an FSM would manage the transitions of

the object from one state to another, such as the door going from closed to opening and open, or open to closing and closed.

In the 2D platformer, the character will have a very simple finite state machine as illustrated in the following diagram:



To elaborate the previous screenshot, the player will start in the **Idle State**. Then, depending on player input or environment interaction, the player will transition into other states, such as the **Walk State** or the **Jump State**. From the walk state, they will either return to the idle state or transition to the jump state, and once in the jump state, they will only be able to return to the idle state. This is a very simple example of a finite state machine, and in many games, characters have dozens of different states. Usually, unique actions are triggered upon entry, update, or exit from a state. In this project, the transitions from one state to another and their updates will be focused on through the use of the **Step** event and designated with user-defined events.

Gathering resources for the platformer

Before the platformer can be started, resources must be created. These assets will focus primarily on the character, which in this case, will be a vampire named Lil' Vlad.

> **Note**
>
> A new project should also be started when creating this new game. Again, a new project can be created by navigating to **File** | **New Project** or pressing *Ctrl* + *N*. This will bring up the **Project** dialog window. This project is simply titled **Platformer**.

Establishing Lil' Vlad's sprites

Relating to the finite state machine mentioned previously, Vlad's character will have three sprites associated with it, as shown in the following list:

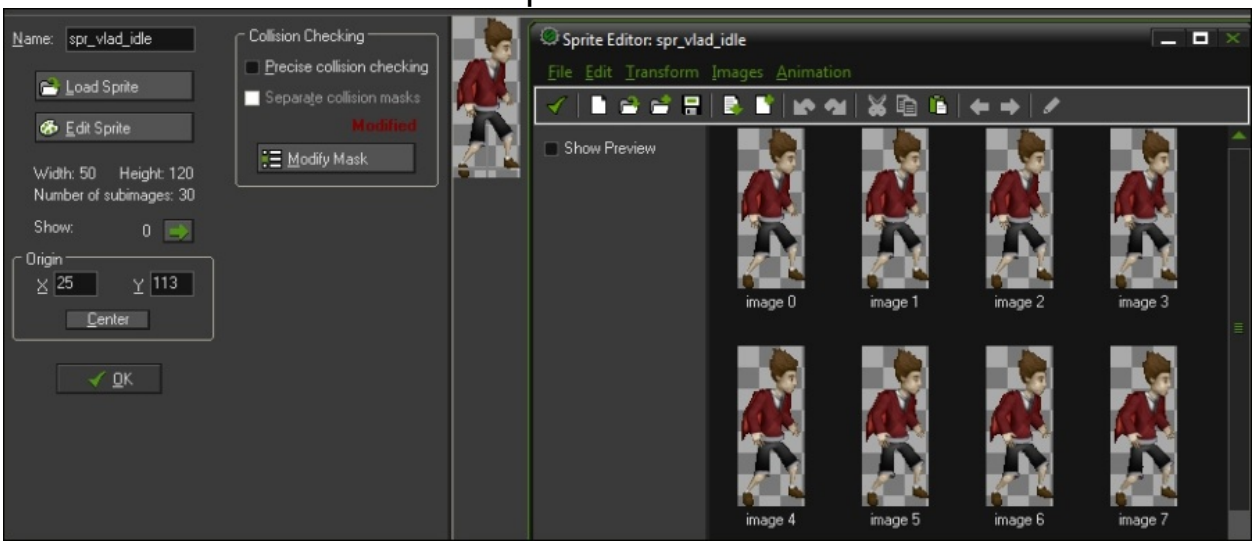
- The `spr_vlad_idle` sprite for the idle state
- The `spr_vlad_walk` sprite for the walking state
- The `spr_vlad_jump` sprite for the jumping state

The standing sprite – spr_vlad_idle

This first sprite resource will be for Vlad's idle animation; it's a 30-frame, looping animation of him standing still. It has several attributes that are important in aiding with positioning and collision.

The first attribute is the origin that is centered horizontally and placed near the bottom of the sprite. The sprite is 50-pixels wide, so the **X** origin is placed at **25**. This is done so that the character can properly be

mirrored when facing different directions. The height of the sprite is 120 pixels. Because the character's position should be where his feet meet the ground, the **Y** origin will be a little higher; in this case, it should be set as **113**. The following screenshot shows the coordinates of the origin as well as several frames of this sprite resource:



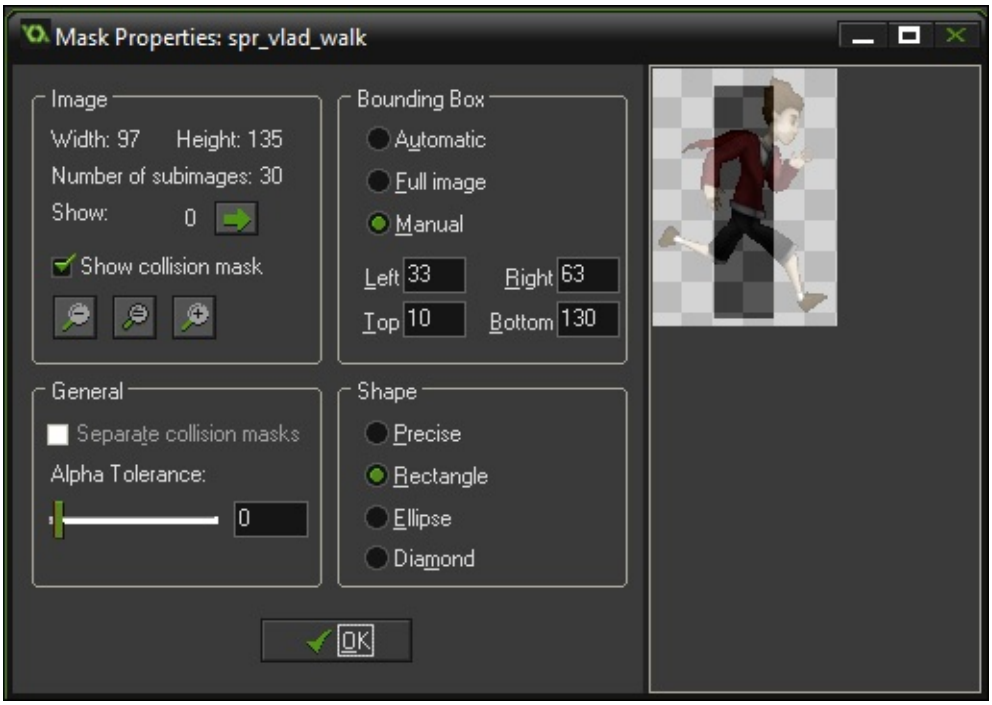
The bounding box is set manually. All bounding boxes used for Vlad will be 30 pixels wide. This setup is done so that the character can collide on either side of the bounding box upon moving through the environment and will be kept consistent for all of the sprites introduced in this chapter. To achieve this, the bounding box should have a value of **10** on the left and **40** on the right. The height for the bounding box will change between states. In the idle state, the bounding box is 108 pixels high, ending just below the tip of the character's spiked hair. The **Bottom** value of the bounding box is the same as the origin, **113**. The **Top** value is then set as **5**. The following screenshot shows this setup:



The walking sprite – spr_vlad_walk

The next sprite resource will be used for Vlad's walk animation. As with the idle animation, it will loop too. Changing the animation of the character at any time serves as feedback to the player indicating that they have entered a new state and should be performing a new action, which is, in this case, horizontal movement. The setup of this sprite is similar to that of `spr_vlad_idle`. The origin is near the bottom at **130**, but it is not quite centered horizontally. The horizontal origin is set to **48**, which is around the character's hips and will allow him to mirror.

The bounding box will be of the same width as `spr_vlad_idle`—30 pixels; the bounding box's left side will be at **33** and the right will be at **63**. The bottom of the bounding box is at the origin, which is **130**. Finally, this sprite is a bit taller than others, the top of it reaching **10**, which makes the bounding box 120 pixels high. This is illustrated in the following

screenshot: 

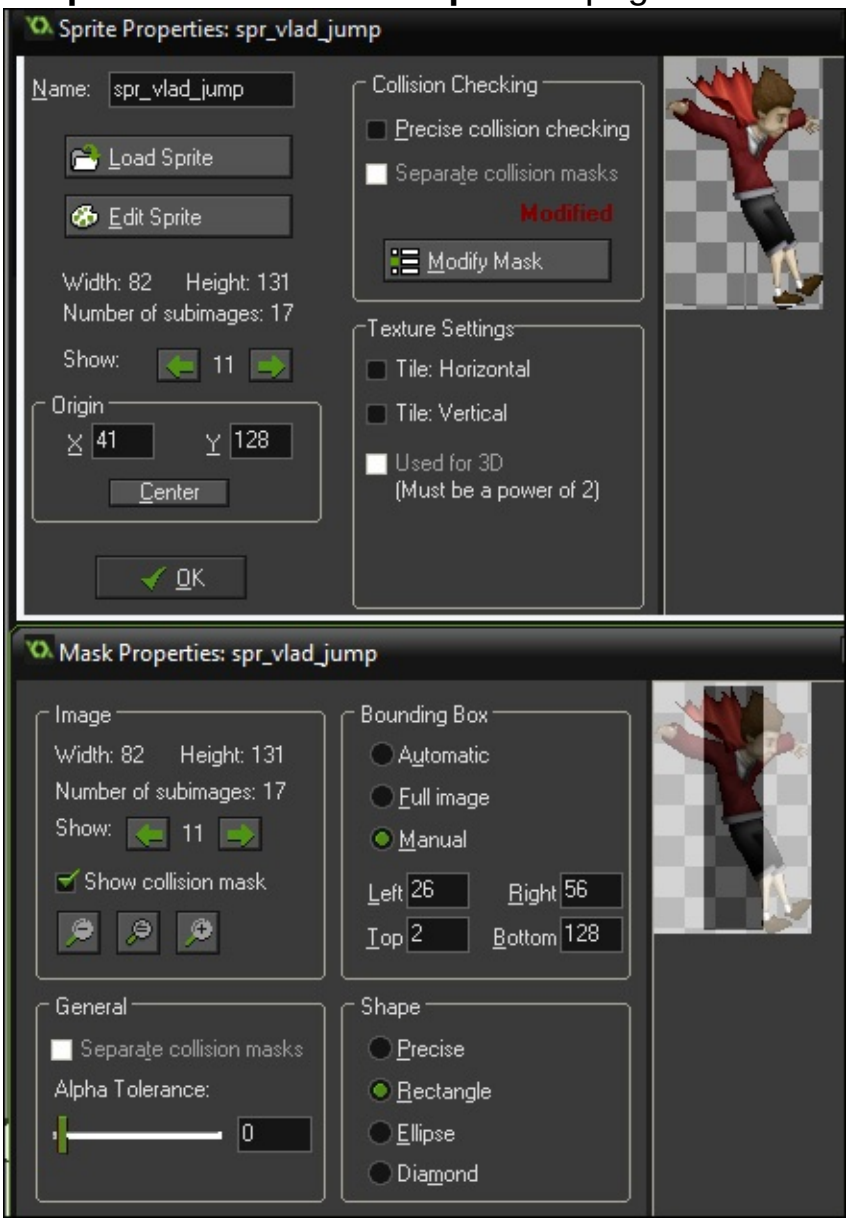
The jumping sprite – spr_vlad_jump

The final sprite resource for this chapter will be used for the animation of Vlad's jump. This will not only be used when the character jumps but also when the character falls off ledges. The first 11 frames account for his ascent and float, while the remaining frames account for his descent. As with the previous two sprites, the origin will be near the bottom and centered at a location where the character mirrors best. The bounding box will then be 30 pixels wide, with 15 pixels on each side of the origin and its bottom at the same point as the origin. The following list contains these different values on the **Sprite Properties** and **Mask Properties** pages:

- The **X** value in **Origin** is set to **41**
- The **Y** value in **Origin** is set to **128**
- The **Left** value in **Bounding Box** is set to **26**
- The **Right** value in **Bounding Box** is set to **56**
- The **Top** value in **Bounding Box** is set to **2**
- The **Bottom** value in **Bounding Box** is set to **128**

The following screenshot shows the use of different values on the **Sprite**

Properties and **Mask Properties** pages:



Jumping with sound – snd_jump

Continuing the trend of giving feedback when possible, a sound resource will be created that will be played when Vlad jumps; this sound will be referred to as `snd_jump`. Jump sounds are usually quick and not grating, as they will be heard quite often.

Creating a new object resource –

obj_vlad

The character, Vlad, will be represented with one object resource named `obj_vlad`. This object resource will utilize six events in total, as shown in the following list:

- The **Create** event: This event is triggered when the character is created
- The **Step** event: This event is triggered in every frame
- The **Animation End** event: This event is triggered when the current animation ends
- Three **User defined** events: These events will be triggered using `event_user` in the **Step** event

The scripts for these events will be defined later in this chapter. The initial sprite assigned should be `spr_vlad_idle`.

Utilizing the User defined events

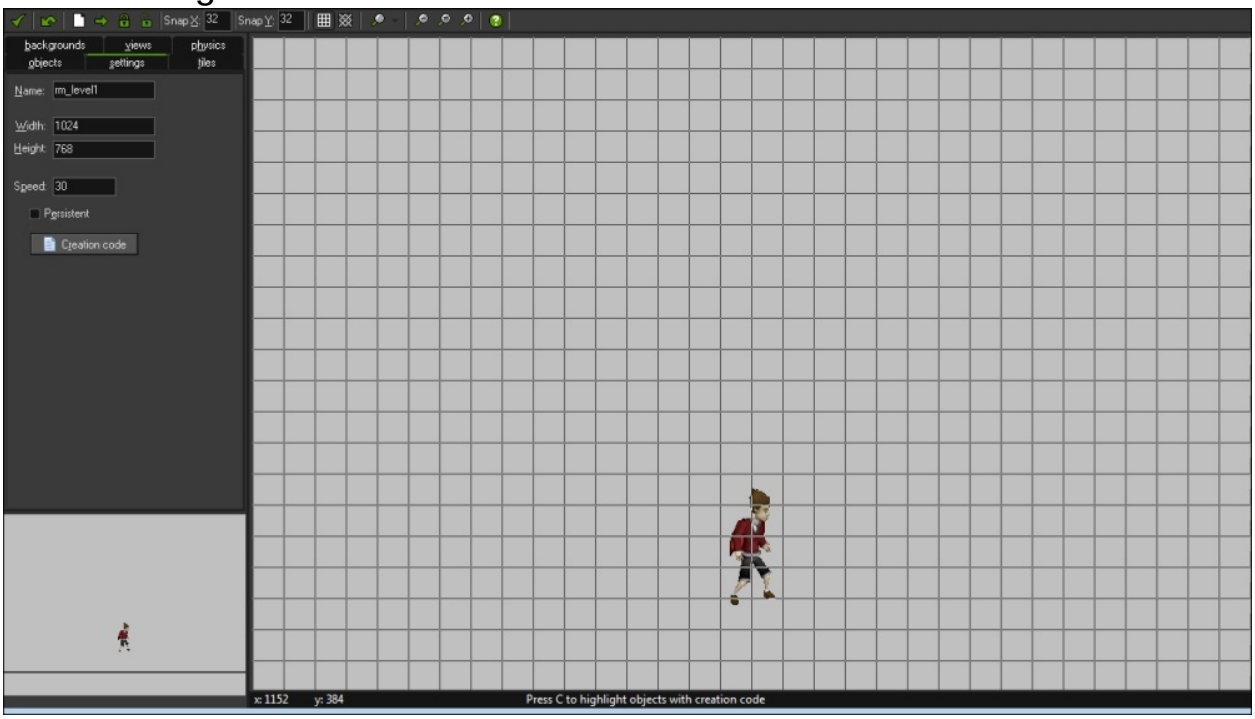
The **User defined** events (**Add Event** | **Other** | **User defined** | **User 0-15**) are custom events that can be triggered through the built-in function `event_user`, whose first argument represents the corresponding index of the desired **User defined** event. Each object resource can have 16 unique **User defined** events.

When updating states, a `switch` statement could be used that tests all the possible states and then performs specific actions based on the current state, but a `switch` statement can become overpopulated very quickly, especially when more complex scripts are written for each state. An array that stores various references to script resources for each state could also be built, but because there will not be more than 16 states, the **User defined** events and `event_user` will be sufficient.

Placing Vlad in a room – rm_level1

A room resource, `rm_level1`, will be created. This room will be **1024** pixels by **768** pixels and contain one instance of `obj_vlad`, as illustrated in

the following screenshot:



The `rm_level1` room will also utilize the **Creation Code** page, which was discussed briefly in Chapter 1, *Getting Started – An Introduction to GML*. The **Creation Code** page is triggered upon the player entering a room. The following is the content in **Creation Code** of `rm_level1`:

```
// Defines the bottom of the room
global.room_bottom = 650;
```

The global variable in the previous code represents an artificial bottom of the room, and will be used when adding jump functionality to the character.

If the resources have been created properly, the game should appear similar to that shown in the following screenshot with Vlad standing idly in the room:



Defining Vlad's state constants

Now that the resources have been created, the states that Vlad will transition to and from should be defined as constants for easier tracking. Again, constants are global variables that cannot be set, and are used to improve code readability for referencing information that is meant to be consistent. The **User-Defined Constants** dialog window can be opened by navigating to **Resources** | **Define Constants...** or pressing *Shift* + *Ctrl* + *N*. The list of name-value pairs will initially be empty. This dialog window has nine buttons (excluding the **OK** button) that are used to populate and edit the following list:

- **Insert**: This button inserts a new constant before the currently selected one
- **Add**: This button creates a new constant at the end of the list
- **Delete**: This button deletes the currently selected constant
- **Clear**: This button removes all defined constants
- **Up**: This button moves the currently selected constant up in the list
- **Down**: This button moves the currently selected constant down in the list
- **Sort**: This button sorts the constants alphabetically by name
- **Load**: This button loads a series of constants from a text file
- **Save**: This button saves the current set of constants as a text file

For this game, three states in the following table will be defined:

Name of the state	Value	Use
`state_idle`	`0`	This state is used to implement the idle state of Vlad
`state_walk`	`1`	This state is used to implement the walking state of Vlad

`state_jump`	2	This state is used to implement the jumping state of Vlad

These states are shown along with the interface for defining constants in the following screenshot:



The values of these constants are very important as they will correlate with the **User defined** events triggered in `obj_vlad`. If created properly, these three values will show up in different colors when editing scripts and in the auto-complete form as well.

Starting Vlad's events and scripts – walking

For the walk state of `obj_vlad`, the player will use the left and right keyboard arrows to move the character. To accomplish this, scripts will be written into the following events:

- The **Create** event
- The **Step** event
- The **User Defined 0** event
- The **User-Defined 1** event

The Create event

Similar to most **Create** events, the **Create** event of `obj_vlad` will handle defining important instanced variables. The following code assigns values to each of these variables that we need to go from the idle state to the walking state and vice versa:

```
/// Variable initialization for character

// The state of the character used within the update
state.
state_id = state_idle;

// Has the player entered a new state.
entered_new_state = false;

// The speed at which the character walks by default.
def_walk_speed = 10;

// The friction used to reduced the character speed
when returning to idle.
def_friction = 1;
```

Right now only four variables are defined. The `state_id` variable is the index of the state that Vlad is currently in, starting with his idle state. The

`entered_new_state` value is a Boolean that signifies whether or not a new state has been entered. The next two variables are prefaced by `def_`, which stands for default. The `def_walk_speed` variable defines the walking speed of the character. Setting this to `10` means that Vlad will travel 10 pixels every frame. Finally, `def_friction` will be used to set Vlad's `friction` value when he is no longer running. The `friction` variable is a built-in variable that, when positive, affects `speed` by making it approach `0` in every frame. In this instance, setting `friction` to `1` will make Vlad's `speed` values—both horizontal and vertical—approach `0` by 1 unit every frame.

> ### Note
>
> When `friction` is negative, the value of `speed` will move away from zero in every frame, which can create rather substantial changes in `speed`.

The Step event

To update Vlad in every frame, a **Step** event will be utilized. The following code will be written into the **Step** event:

```
/// State machine update.

// Gets the current state id.
var current_state = state_id;

// Updates the event.
event_user(state_id);

// If the event has changed, the entry of a new state
is designated.
if (current_state != state_id)
{
    entered_new_state = true;
}
```

In the first portion of this code, the current value of `state_id` is assigned

to a local variable. Then, one of the **User defined** events is triggered based on the state. Finally, `state_id` and `current_state` are compared; if they differ, which means that the state has changed while it was being updated, `entered_new_state` is set to `true`.

Standing still – the User Defined 0 event

The idle state is a rather important state; it is the starting state and can transition to most other states. In this case, the keyboard will be used to change the state so that the character can move left and right. Also, the use of `entered_new_state` will be demonstrated. The following is the code for the idle state:

```
/// IDLE STATE

// If the idle state has been entered.
if (entered_new_state)
{
    // Sets the sprite index, image index, and loop
index.
    sprite_index = spr_vlad_idle;
    image_index = 0;

    // Sets the friction
    friction = def_friction;

    // Set to false so the entry functions are not
called again.
    entered_new_state = false;
}

// If left arrow is down...
if (keyboard_check(vk_left))
{
    // Changes the state id.
    state_id = state_walk;

    // Mirrors the character so it is facing left.
    image_xscale = -1;
}
// If right arrow is down.
else if (keyboard_check(vk_right))
{
```

```
    // Changes the state id.
    state_id = state_walk;

    // Sets the x scale to 1 to make sure the
character is facing right.
    image_xscale = 1;
}
```

In the previous block of code, the first `if` statement checks `entered_new_state`. If `entered_new_state` returns `true`, several actions are performed. Firstly, the `sprite_index` value is set to the idle animation, and `image_index` is set to start at the frame `0`. Then, the friction is set to `def_friction`. This is done to give the character a nice, smooth stop as opposed to a jarring one when returning from the walk state. Finally, `entered_new_state` is set to `false` so that it does not continually execute at every update.

After this, the keyboard is checked using `keyboard_check`. If the left arrow key is held down, `state_id` is set to the predefined constant `state_walk`, and the character is mirrored to look left by setting `image_xscale` to `-1`. If the left arrow key is not held down, but the right one is, the state is still set to `state_walk`, but `image_xscale` is set to `1`.

Walk this way – the User Defined 1 event

If the game were to be tested now, nothing would happen if the right arrow key is pressed, and if the left arrow key is pressed, Vlad would simply mirror and face left. Behind the scenes, he would have entered the walk state, but because the **User defined** event has not been coded, he would get locked into the walk state and be unable to return to the idle state. By writing the following script in the **User Defined 1** event, Vlad will be able to enter and exit the walk state:

```
/// WALK STATE

// If the walk state has been entered.
if (entered_new_state)
{
    // Sets the sprite index, image index.
```

```
    sprite_index = spr_vlad_walk;
    image_index = 0;

    // Sets the friction and horizontal speed,
reversing it if the character is facing left.
    friction = 0;
    hspeed = def_walk_speed;
    if (image_xscale < 0)
    {
        hspeed *= -1;
    }

    // Set to false so the entry functions are not
called again.
    entered_new_state = false;
}

// If the left arrow is released while facing left or
the right arrow is released while facing right...
if ((image_xscale < 0 && !keyboard_check(vk_left)) ||
(image_xscale > 0 && !keyboard_check(vk_right)))
{
    // Return to the idle state.
    state_id = state_idle;
}
```

Similar to the **User Defined 0** event, the first `if` statement checks whether or not Vlad has recently entered the walk state. If he has, the `sprite_index` object must be changed. Then, `friction` is set to `0` so that it doesn't affect `hspeed`, which is set to `def_walk_speed`. If the value of `image_xscale` is less than `0`, meaning the character is facing to the left, the `hspeed` value is multiplied by `-1`. Finally, `entered_new_state` is set to `false` to prevent it from continually updating.

Then, a compound Boolean is checked to determine when Vlad should re-enter the idle state. If Vlad is facing to the left and the left key isn't being held down anymore, or he is facing to the right and the right arrow key isn't being held down, he should return to the idle state.

If these scripts are entered properly, Vlad should be able to run to left and right in the scene, slowing down slightly upon returning to the idle state. In the next section, the scripts of the **Create** and **Step** events will

be amended and the **User Defined 2** event and the **Animation End** event will be defined so that Vlad can jump!

Adding new variables for jumping

To make Vlad jump, more variables must be defined. These variables are associated with the default parameters for making Vlad jump, such as `jump_speed` and `gravity`. These variables will be scripted into the **Create** event as shown in the following code:

```
// The speed applied to the character when jumping by
default.
def_jump_speed = -15;

// The gravity applied to the character by default.
def_gravity = 1;

// The number of times the character has jumped.
jump_count = 0;

// The number of times the character is allowed to
jump
jump_limit = 2;

// Is the character falling; used to differentiate
jumping from falling.
grounded = false;

// At which frame does the animation loop?
loop_index = 0;
```

The `def_jump_speed` variable represents the initial jump speed that Vlad's `vspeed` value will be set to when entering the jump state; meanwhile, `def_gravity` is the value that will set the built-in, instanced variable `gravity`. Using a value of `-15` for `def_jump_speed` means that Vlad will start moving up at a rate of 15 pixels per frame. During every frame, the `speed` value will be increased by `gravity`, which is set to `1`, eventually causing the character to fall back down.

The `jump_count` variable represents the number of times the character has jumped. The `jump_limit` variable represents the number of times

Vlad is allowed to jump. Defining `jump_limit` is useful to allow skills such as double jumping, and could even be an unlocked ability; its value can start out as `1` and then, with the use of a special item, be increased to reach new sections in a level. This is jumping ahead a bit, so for now, `jump_limit` will be set to `2`, which will allow Vlad to double jump.

The `grounded` variable is used to represent whether or not Vlad's feet are "planted". It will also be used to determine whether or not Vlad is actually jumping or falling.

Finally, `loop_index` is a value that will be applied to `image_index`, making Vlad's animation loop correctly when performing the jump animation since it can look rather strange otherwise.

Using up to jump – the Step event update

In this section, code will be added to the **Step** event so that Vlad can jump. Vlad can enter the jump state from both the idle and the walking states, so adding the following code to both events is redundant, and instead, is added to the **Step** event. This update, as shown in the following code, should be scripted after `event_user` is called, but before `current_state` and `state_id` are compared:

```
/// State machine update.

// Gets the current state id.
var current_state = state_id;

// Updates the event.
event_user(state_id);

// Jump functionality, which can be performed in
either state except jump.
if (state_id != state_jump)
{
    // If above the bottom of the room, the character
falls.
    if (y < global.room_bottom && place_empty(x, y +
1))
    {
        state_id = state_jump;
```

```
            grounded = false;
        }

        // If grounded, the jump count is in range, and
    the spacebar was pressed, the character will jump.
        if (grounded && jump_count < jump_limit &&
    keyboard_check_pressed(vk_up))
        {
            state_id = state_jump;
        }
    }

    // If the event has changed, the entry of a new state
    is designated.
    if (current_state != state_id)
    {
        entered_new_state = true;
    }
```

An `if` statement is used first to make sure this check only occurs when Vlad is not in the jump state. Then `global.room_bottom`, which is defined within the **Creation Code** page of the room, is compared to the current position. If Vlad is above it, the jump state is entered; it is also designated that the character is not grounded. Meanwhile, if the character is grounded, the number of jumps performed is less than the specified limit, and the up arrow has been pressed, the state is changed to `state_jump`.

Falling state – the User Defined 2 event

The next script will execute several actions when Vlad enters the jump state. Jumping is a much more complex action than walking and standing, and small touches have been added to make jumping feel more responsive and give the player more control, similar to how `friction` is applied when returning to the idle state, as shown in the following code:

```
    /// JUMP  FALL STATE

    / If the state was newly entered.
    if (entered_new_state)
    {
        // Sets sprite index and looping index.
        sprite_index = spr_vlad_jump;
```

```
    loop_index = 11;

    // If grounded, the vertical speed is set to
perform another jump and the jump counter is
incremented.
    if (grounded)
    {
        vspeed = def_jump_speed;
        image_index = 0;

        audio_play_sound(snd_jump, 0, false);
    }
    else
    {
        image_index = 11;
    }

    // Jump counter is incremented, even for falling.
    jump_count++;

    // Sets the gravity and friction.
    gravity = def_gravity;
    friction = 0;

    // Designates that the character is not grounded.
    grounded = false;

    // Set to false so the entry functions are not
called again.
    entered_new_state = false;
}
```

The initial portion of this script deals with entering the jump state. The `image_index` and `loop_index` variables are set to `11` since the descending loop cycle begins at frame 11. Now, the variable `grounded` is used. If the character is grounded, `vspeed` is set to `def_jump_speed`, the animation is set to start at `image_index` with the value `0`, and the jump sound that was created earlier is played. If the character is not grounded, `image_index` is simply set to the same frame as `loop_index`. This is followed by incrementing the value of `jump_count`, setting `gravity` and `friction` to their designated default values, and changing `grounded` as well as `entered_new_state` to `false`. The code in the next section will handle changing the character's speed with left and right key input:

```
// If the left or right arrow key is pressed, the
character moves slightly in that direction...
if (keyboard_check(vk_left))
{
    image_xscale = -1;
    hspeed = max(hspeed - def_friction, -
def_walk_speed);
}
else if (keyboard_check(vk_right))
{
    image_xscale = 1;
    hspeed = min(hspeed + def_friction,
def_walk_speed);
}
// otherwise the character returns to a neutral,
horizontal speed.
else
{
    if (hspeed < 0)
    {
        hspeed = min(hspeed + def_friction, 0);
    }
    else if (hspeed > 0)
    {
        hspeed = max(hspeed - def_friction, 0);
    }
}
```

In the previous code, if the left arrow key is held down, Vlad will face to the left and approach the negative of the default speed at the rate of the predefined `friction` variable. If the right arrow key is held down, the opposite is done. Finally, if neither key is held down, the horizontal speed approaches zero. The `hspeed` value is compared with `def_friction` instead of `0` to prevent a teetering effect in case the addition or subtraction of the `hspeed` value results in the value going below or above `0` and not settling on it. These calculations must be performed because `friction` cannot be set; doing this would affect the vertical speed as well.

If these keyboard checks were omitted, the character's jumping speed would be determined by the previous state. If the character is in the idle state, he would jump vertically and if he is in the walking state, he would

jump with the same speed and in the direction he is already facing. This is rather basic, but it's nice to be able to make the character change direction or slow down for more controlled jumps. At the same time, changing direction in the air shouldn't be as fast as it would be when the character is on the ground, so when buttons associated with `vk_left` or `vk_right` are pressed, he adjusts the trajectory of his jump slightly instead of immediately.

Another bit of polish that can be added to the jump is the ability to make the character not always jump at the same height. Also, an action to test for double (or any additional number of) jumps should be performed. Scripting the following will allow this to happen:

```
// If the character is moving up, and the up arrow is
released, the speed is divided in half...
if (vspeed < 0 && keyboard_check_released(vk_up))
{
    vspeed *= 0.5;
}
// If the jump count is less than the limit and the up
arrow is        pressed...
else if (jump_count < jump_limit &&
keyboard_check_pressed(vk_up))
{
    // Set to true to renter the jump state.
    entered_new_state = true;

    // Sets grounded to true so the character jumps
and doesn't fall.
    grounded = true;
}
```

So first, `vspeed` is checked and if its value is less than `0`, which means the character is ascending, and the up arrow is released, the value of `vspeed` is multiplied by `0.5`. This will give the illusion of the jump ending early and smoothly instead of just immediately stopping in midair and reversing direction. However, if the up arrow is pressed while the value of `jump_count` is less than that of `jump_limit`, the `entered_new_state` and `grounded` variables are both set to `true` so that on next update, another jump is performed.

The following final section will define what happens when the character reaches `global.room_bottom`:

```
// If the y position plus the vertical speed is below
the bottom of   the room...
if (y + vspeed > global.room_bottom)
{
    // Character is moved to the bottom of the room.
    y = global.room_bottom;

    // Speed and gravity are set to 0 to stop vertical
movement.
    vspeed = 0;
    gravity = 0;

    // Jump counter is reset.
    jump_count = 0;

    // The state is set to idle.
    state_id = state_idle;
}
```

The final portion of code helps end the jump. If the sum of Vlad's `y` position and his vertical speed is greater than the value of the bottom of the room defined in the **Creation Code** script of `rm_level1`, he will enter the idle state. This prediction is made so that Vlad's position doesn't fall past this value for one frame and then snap back to the position he should settle at in the next. Before entering though, the `vspeed` and `gravity` values are both set to `0`, and the `y` coordinate is set to `global.room_bottom`. The `jump_count` value is also set to `0` so that Vlad can jump again, now that he has returned to the ground. This is a very important step; if this were not done, Vlad would fall offscreen and into a virtual abyss forever or until the game is stopped.

The code for the jump state is rather complex and lengthy. For clarification, the following is the script in its entirety:

```
/// JUMP  FALL STATE

/ If the state was newly entered.
if (entered_new_state)
```

```
{
    // Sets sprite index and looping index.
    sprite_index = spr_vlad_jump;
    loop_index = 11;


    // If grounded, the vertical speed is set to
perform another jump and the jump counter is
incremented.
    if (grounded)
    {
        vspeed = def_jump_speed;
        image_index = 0;

        audio_play_sound(snd_jump, 0, false);
    }
    else
    {
        image_index = 11;
    }

    // Jump counter is incremented, even for falling.
    jump_count++;

    // Sets the gravity and friction.
    gravity = def_gravity;
    friction = 0;

    // Designates that the character is not grounded.
    grounded = false;

    // Set to false so the entry functions are not
called again.
    entered_new_state = false;
}

// If the left or right arrow key is pressed, the
character moves slightly in that direction...
if (keyboard_check(vk_left))
{
    image_xscale = -1;
    hspeed = max(hspeed - def_friction, -
def_walk_speed);
}
else if (keyboard_check(vk_right))
{
    image_xscale = 1;
```

```
    hspeed = min(hspeed + def_friction,
def_walk_speed);
}
// otherwise the character returns to a neutral,
horizontal speed.
else
{
    if (hspeed < def_friction)
    {
        hspeed = min(hspeed + def_friction, 0);
    }
    else if (hspeed > def_friction)
    {
        hspeed = max(hspeed - def_friction, 0);
    }
    else
    {
        hspeed = 0;
    }

}

// If the character is moving up, and the spacebar is
released, the speed is divided in half...
if (vspeed < 0 && keyboard_check_released(vk_up))
{
    vspeed *= 0.5;
}
// If the jump count is less than the limit and the
spacebar is pressed...
else if (jump_count < jump_limit &&
keyboard_check_pressed(vk_up))
{
    // Set to true to renter the jump state.
    entered_new_state = true;

    // Sets grounded to true so the character jumps
and doesn't fall.
    grounded = true;
}

// If the y position plus the vertical speed is below
the bottom of the room...
if (y + vspeed > global.room_bottom)
{
    // Character is moved to the bottom of the room.
    y = global.room_bottom;
```

```
    // Speed and gravity are set to 0 to stop vertical
movement.
    vspeed = 0;
    gravity = 0;

    // Jump counter is reset.
    jump_count = 0;

    // The state is set to idle.
    state_id = state_idle;
}
```

If the game were to be played now, Vlad's three states should almost be complete except for one minor detail. When Vlad is jumping, it is noticeable that about halfway through the jump, he re-enters the jump animation from the beginning, which creates a rather confusing effect. In the next section, this will be remedied with the **Animation End** event and minor updates to the idle and walk states events.

Looping the jump – the Animation End event

Until this point, the variable `loop_index` has not been utilized. The **Animation End** event (**Add Event** | **Other** | **Animation End**), which is triggered whenever the final frame of an animation is reached, will use `loop_index` now to prevent Vlad's jump animation from starting over midjump, as shown in the following code:

```
/// Loops the animation to the designated frame upon
completion of the animation.
image_index = loop_index;
```

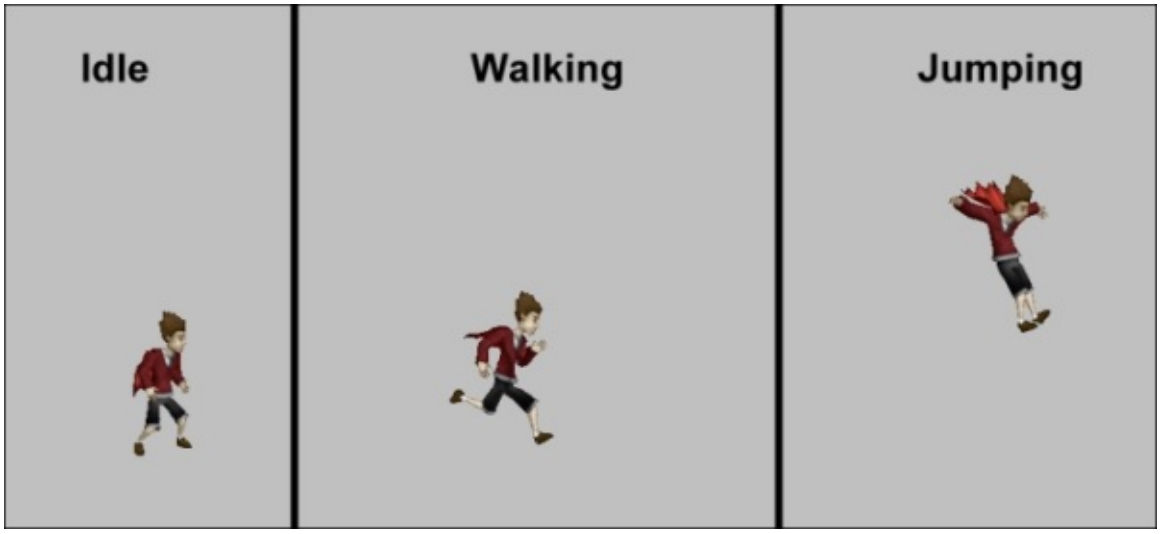
It can be said that this code is rather simple. It's handling a minor detail, but these details can sometimes help resolve or create confusion while playing that otherwise would be present. Before continuing, `loop_index` must be assigned when the idle and walk states are entered. In the **User Defined 0** event, the following code can be added:

```
// If the idle state has been entered.
if (entered_new_state)
{
    // Sets the sprite index, image index, and loop
index.
    sprite_index = spr_vlad_idle;
    image_index = 0;
    loop_index = 0;
```

As shown in the following code, in the **User Defined 1** event, `loop_index` should be assigned the value `0` similar to the previous code:

```
// If the walk state has been entered.
if (entered_new_state)
{
    // Sets the sprite index, image index, and loop
index.
    sprite_index = spr_vlad_walk;
    image_index = 0;
    loop_index = 0;
```

Now that the previous code has been written, Vlad's animation should loop properly in all three states and Vlad should be able to run and jump around the room in those states, just as shown in the following screenshot:



Summary

In this chapter, a new game was created starring a little vampiric character named Vlad. Code was written so that this character, using a finite state machine, could transition between three states: standing idle, walking, and jumping. Scripts were also written so the character could transition between these three states through the use of the **User defined** events, the **Step** event, and keyboard input. Some of GameMaker: Studio's built-in movement variables, such as `hspeed`, `friction`, and `gravity`, were also discussed. Getting the character basics down early is a very important step in building a successful platformer.

Now, character movement and interaction are nice, but the game is missing its most important asset: platforms! In the next chapter, collision will be discussed so that platforms can be created. This will give Vlad elements that he can actually walk and jump onto!

Chapter 7. It's in the Name – Platforms and Collisions

In the previous chapter, we began the construction of a platformer game. So far, the game's main character, Vlad, can run and jump through the use of a finite state machine. Up to this point, however, there hasn't been anything for him to actually run and jump on! In this chapter, platforms will be created that he'll be able to interact with.

First, the concept of collision and the functions available to work with collision will be explained. Then, three types of platforms will be created as shown in the following list:

- A solid platform
- A platform that Vlad can walk in front of but still jump on top of
- A platform that will be moved through the use of path resources

Collision – a crash course

Before creating assets and platforms for Vlad to jump on, the concept of collision should be discussed. Collision in this book refers to when two objects intersect and the reactions that occur based on this intersection.

Creating masks for collision

A plethora of complex geometric equations and programmatic functions exists to determine the intersection between various shapes such as rectangles and circles. Fortunately, these are taken care of by GameMaker: Studio. In GameMaker: Studio, **masks**, which have been briefly discussed in earlier chapters, are used to calculate the collision between objects as well as register certain **Mouse** events. Different shapes can be chosen to define a mask through the **Mask Properties** dialog window, which can be accessed from the **Sprite Properties** window by clicking on the **Modify Mask** button. The options for shapes

available are as follows:

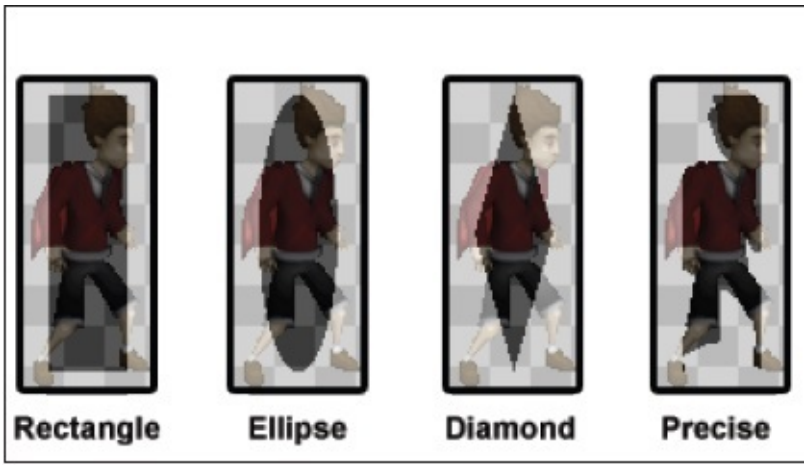
- **Rectangle**: This option creates a rectangular bounding box
- **Ellipse**: This option creates a rounded bounding box
- **Diamond**: This option creates a diamond-shaped bounding box
- **Precise**: This option uses the alpha of the sprites to determine the collision

When working with precise collision, there is a checkbox referring to the separation of the collision masks in a sprite. If this is unchecked, an accumulative precision mask will be constructed using all subimages of the sprite; otherwise, the precise collision mask will change based on the current image index.

Note

Precise masks, though very accurate, can be very processor intensive, especially if a lot of objects utilize them. In platformers, they can also cause issues as they get stuck on the edges. They are best used sparingly.

When defining a mask, the **Top**, **Left**, **Right**, and **Bottom** values are still used, but the resulting shapes are different. These four shapes are demonstrated as follows using `spr_vlad_idle` as a representation:



Working with placement and movement functions

Two types of functions exist that are very helpful when working with collision: those that test the position of the object and those that move the object. This section will cover these two types of functions and explain their use more thoroughly.

> **Note**
>
> Not all of these functions deal with collision but they can be very useful nonetheless.

Testing placement

The following functions test to see if the instance that executes them meets certain requirements:

- `place_empty(x, y)`: This function checks to see whether there will be a collision with any other instance if the current instance is moved to the designated `x` and `y` positions
- `place_free(x, y)`: This function checks to see whether there will be a collision with any other solid instances if the current instance is moved to the designated `x` and `y` positions
- `place_meeting(x, y, obj)`: This function checks to see whether there is collision with the object instance specified in the third argument if the current instance is moved to the designated `x` and `y` positions
- `place_snapped(horizontal_snap, vertical_snap)`: This function checks to see if the object's current position is within the grid of the specified values

Movement functions

Movement functions will move and update objects based on certain parameters. Some will change the position of the object, while others may affect speed. The following are a few movement functions:

- `move_towards_point(x, y, speed)`: This function sets the speed and direction of the instance so that it approaches the designated point. It is useful for making one object follow another.
- `move_wrap(wrap_horizontal, wrap_vertical, margin)`: This function wraps the instance around the room. The first argument is a Boolean that specifies if the instance should wrap horizontally, the second Boolean argument specifies if the instance should wrap vertically, and the third value represents how many pixels outside the room the instance must be before wrapping.
- `move_random(horizontal_snap, vertical_snap)`: This function moves the object to a random position within an invisible grid.
- `move_snap(horizontal_snap, vertical_snap)`: This function moves the instance to the nearest grid positions. A good use of this would be after an execution of `place_snapped` returns `false`.
- `move_contact_all(direction, max_distance)`: This function moves an object in a specified direction until it comes in contact with another object. The first argument is the angle to be tested in degrees. The second argument gives how many pixels the object should test to complete this move. If `0` is provided, 1,000 pixels will be tested. This function is often used to move an object directly next to another.
- `move_contact_solid(direction, max_distance)`: This function is similar to `move_contact_all`, except it will try to make contact with only solid objects.
- `move_outside_all(direction, max_distance)`: This function moves an object in a specified direction until it will no longer come in contact with any object. The arguments for this function are the same as `move_contact_all`.
- `move_outside_solid(direction, max_distance)`: This function is similar to `move_outside_all`, but it only tests against solid objects.
- `move_bounce_all(use_precision)`: This function makes the instance bounce off any object if a collision occurs by reflecting its speed and direction. Its sole argument is a Boolean that specifies whether or not precise collision should be used. Using precise collision,

especially if there are many instances in the room, can be rather processor intensive and should be used with caution.

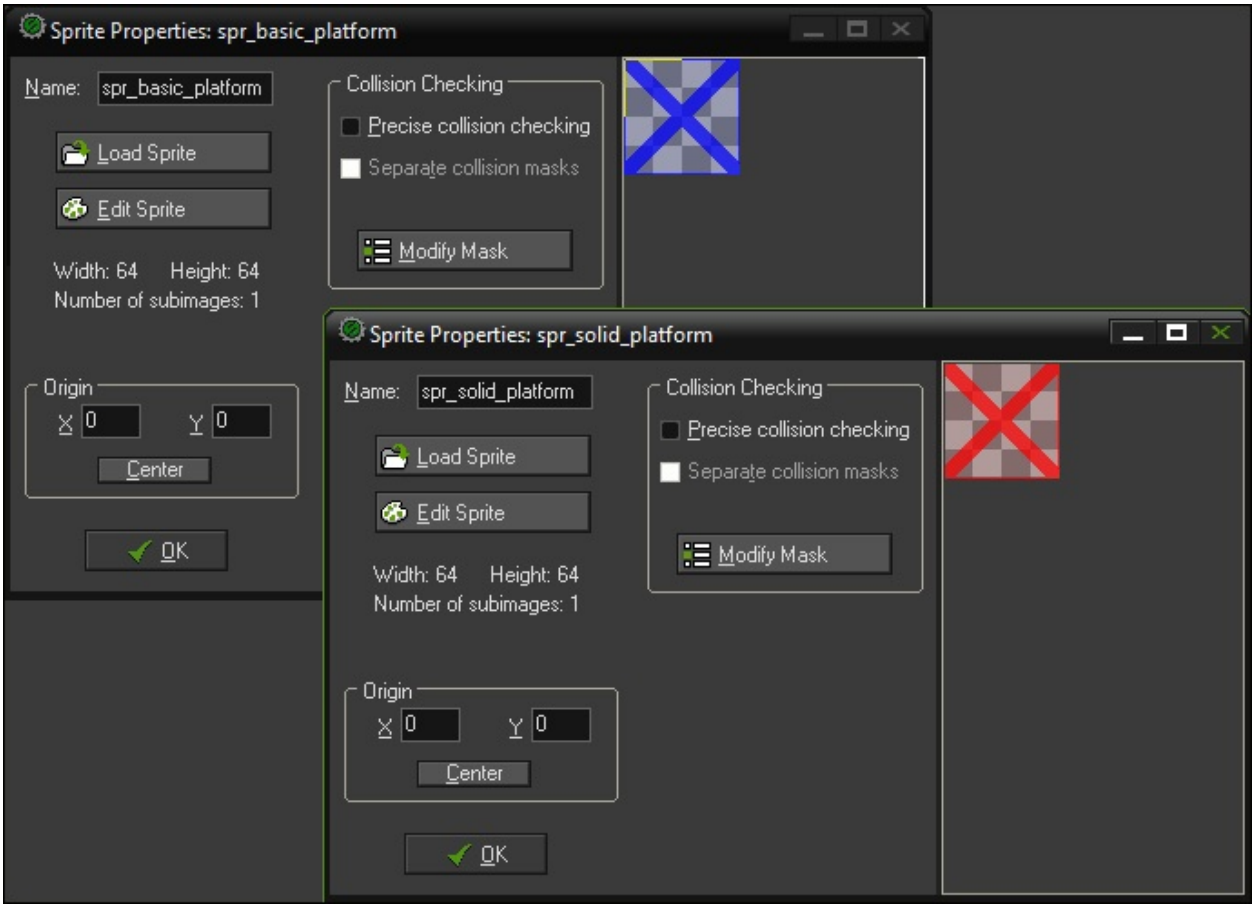
- `move_bounce_solid(use_precision)`: This function is similar to `move_bounce_all`, but it only tests against solid objects.

Gathering resources to build platforms

Now that the basic functions to test and interact with collision have been introduced, the resources needed to integrate platforms into this game can be made. There will be two types of stationary platforms: those that Vlad can land only on top of and those that are completely impenetrable.

Sprites – spr_basic_platform and spr_solid_platform

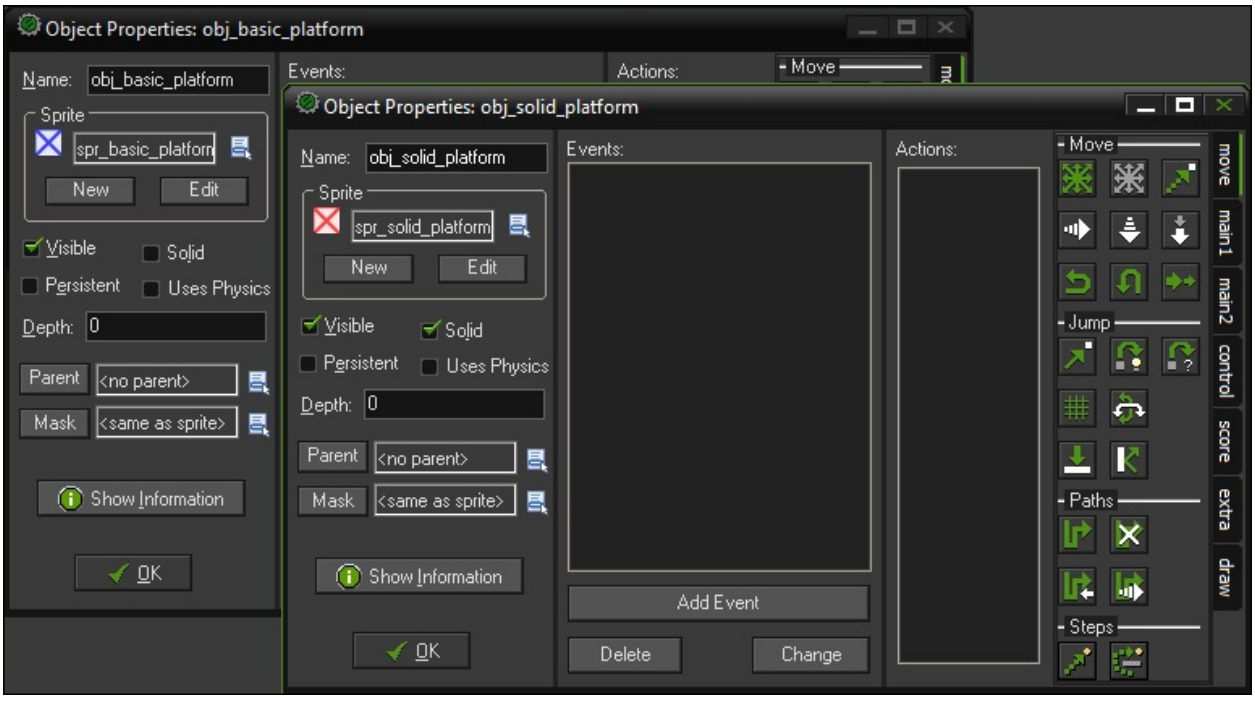
These two sprite resources are very similar, and they are used to help differentiate the two types of platforms in the game; this is illustrated in the following screenshot:



These two sprite resources' origins are not modified, leaving the origin in the upper-left corner at (0, 0). Their masks are also not modified and will encompass the entire sprite. The sprites are semitransparent with an X to help show intersection between platforms when populating the room.

Objects – obj_basic_platform and obj_solid_platform

The objects that use the previously created sprites are rather simple as well. The following screenshot shows their parameters that are identical, except for two things: they use different sprites, and the **Solid** parameter of `obj_platform_solid` is checked. The following screenshot shows the **Object Properties** window:



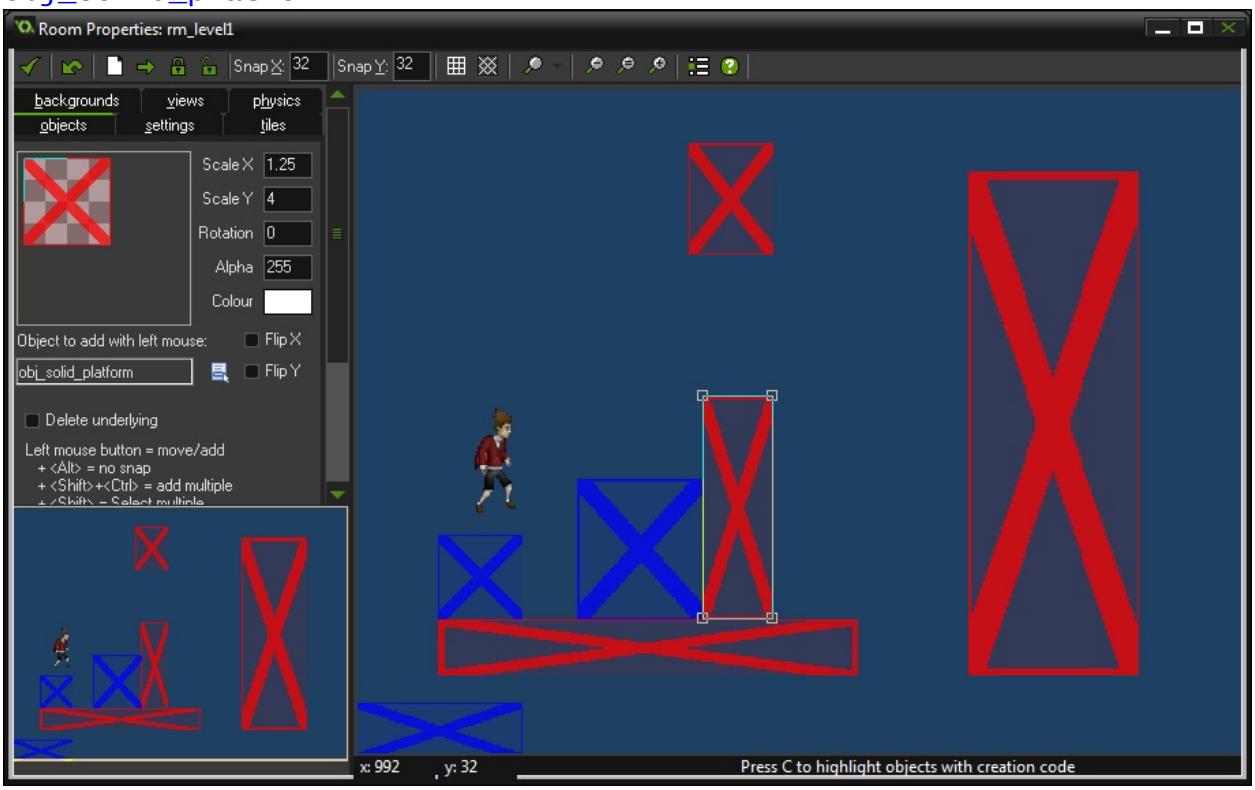
Solidifying objects

Previously, the object resource `obj_solid_platform` was marked as `solid`. This is a feature built into GameMaker: Studio. The goal of designating an object as solid is to make the object impenetrable and prevent visible intersection between it and any other object upon

collision. Also, object resources marked as solid should not be moved as they can cause collision problems, such as objects getting stuck inside each other.

Populating the room

The two newly created platforms can now be placed into the room, `rm_level1`, created in the previous chapter. The layout of this room really comes down to level design at this point. The following screenshot shows a simple layout that uses both `obj_basic_platform` and `obj_solid_platform`:



In the previous screenshot, instances of `obj_basic_platform` and `obj_solid_platform` are scaled instead of repeated. The parameters, such as **Scale X**, **Scale Y**, **Rotation**, **Alpha**, and **Color**, of a selected object instance can be accessed from the **objects** tab of the **Room Properties** window. Though it normally would take thousands of objects to really cause performance issues, having fewer objects is usually preferred when possible.

Now if the game is played, there will be several issues. One is that the collisions between the character and the instances of the `obj_basic_platform` object will have no reaction. The other is that when

landing onto instances of `obj_solid_platform`, Vlad will not leave the falling state, and if left alone, will eventually fall through the collision. This is because no **Collision** events have been created! These events will be added to `obj_vlad` in the next section.

Working with Collision events

Collision events are triggered when one object collides with another. These events can be added to any object. When adding a **Collision** event in the **Choose the Event to Add** dialog window, a list of all the currently available object resources will appear. The newly created event will then be triggered when the object being modified collides with any instance of that object.

Updating the Create event of obj_vlad

Before writing scripts to handle collision, the addition of an instanced variable is required. This should be added to the script in the **Create** event of `obj_vlad` as follows:

```
// Tolerance for how many pixels below a platform the
character can still be moved to the top.
step_threshold = 16;
```

This newly created variable, `step_threshold`, will act as a tolerance buffer. If the character is 16 units below the top of a platform, they will still move to the top of it. It can be useful for creating steps or stairs, allowing the character to climb steps shorter than `step_threshold`.

The Collision events of obj_vlad

In `obj_vlad`, two collision events should be created. One for colliding with `obj_solid_platform` and one for `obj_basic_platform`. The code in both events is identical, as shown in the following code snippet:

```
/// Runs the collision script.
scr_test_collision(other);
```

In the code, the previously created script resource `scr_test_collision` was used, and it supplied one variable, `other`. The variable `other` is a built-in variable that represents the `id` value of the object that collided

with the instance that is triggering the **Collision** event. Individual scripts could have been written within these events to account for the collision of these two objects differently, but it's much more efficient to write only one.

Creating the script resource scr_test_collision

This is a new script resource that should be created and will be utilized in the **Collision** events triggered when `obj_vlad` collides with instances of `obj_basic_platform` and/or `obj_solid_platform`. The goals of this script are as follows:

- Use solid collision to act as walls that block Vlad's path
- Allow Vlad to land on top of both the solid and nonsolid collisions and go into his idle state upon doing so
- Interrupt Vlad's jump if he collides with a solid collision from below

In this script, the two types of possible collisions—horizontal and vertical—must be checked. The following first section accounts for horizontal collision:

```
if (argument0.solid && !place_free(x + hspeed, y + 1))
{
    // If the character would collide, it is moved
back by its hspeed.
    x -= hspeed;
}
```

In the previous code, the solidity of `argument0`, where `argument0` represents the collided object, is checked. If it's not solid, then the walls are not being checked; thus, a check between `obj_vlad` and this object is not required. If the object that collided with the wall is solid, however, `place_free` is utilized using the sum of the current `x` position and horizontal speed and the character's `y` position plus 1.

So, in addition to checking whether or not the collided object is solid, a predictive test is carried out testing that, if the character were to move,

would it still collide with a solid object or would it be free. If it were to collide, the character would be moved back by its horizontal speed. The predictive test is carried out because, by default, objects are already able to move out of solid collisions. This, however, can sometimes make characters appear to be "locked" into a collision if the speed is not accounted for.

In the following second portion of this script, the collision is checked vertically:

```
// If the the character collides vertically with the
object.
if (place_meeting(x + hspeed, y + vspeed, argument0))
{
    // If the character is moving down and above the
bottom of the platform...
    if (y - vspeed < argument0.y)
    {
        // Moves the character to the top of the
platform.
        y = other.y - 1;

        // Vertical speed and gravity are reduced to 0
to prevent continuous falling.
        vspeed = 0;
        gravity = 0;

        // Enter the idle state and reset the jump
counter.
        state_id = state_idle;
        entered_new_state = true;
        jump_count = 0;

        // Assigns the current platform
        current_platform = argument0.id;
    }
    // If the character hits a solid collision from
the bottom.
    else if (argument0.solid)
    {
        // Moves the character in reverse of the y
position,similarly to the x-position.
        y -= vspeed;
```

```
        // Speed is set to 0 to act like they were
stopped.
        vspeed = 0;
    }
}
```

The predictive check is done again both horizontally and vertically, but just on the object that has collided with the player. It's important to still check horizontally because if the first horizontal collision test returned `false`, the character could be colliding with an object when this new test for vertical collision was performed. Then, if the position of the character minus its vertical speed is less than the sum of the collision's `y` and `step_threshold` values, the character will land on this collision.

To complete this landing process, the character will snap to the top of the position minus `1` to prevent continuous collision. This works because the origin of each platform is in its upper-left corner. Then, the vertical speed and gravity are both set to `0` to prevent a continuous fall. The `state_id` variable is set to `state_idle`—a constant defined in the previous chapter—and `entered_new_state` is set to `true` to make Vlad go into his idle state. Finally, `jump_count` is set to `0` so that he can continue to jump.

On the other hand, if the other object is solid, the character will be moved back by the vertical speed, and this speed will then be set to `0`. This will make it as if the character hit his head and is being stopped by the previous collision.

These two collision checks involve some rather complex testing, so the following is the script in its entirety:

```
/// Performs platform collision testing.

// argument0 -- the platform collided with.

if (argument0.solid && !place_free(x + hspeed, y + 1))
{
    // If the character would collide, it is moved
back by its hspeed.
    x -= hspeed;
}
```

```
// If the the character collides vertically with the
object.
if (place_meeting(x + hspeed, y + vspeed, argument0))
{
    // If the character is moving down and above the
bottom of the platform...
    if (vspeed > 0 && y - vspeed < argument0.y +
step_threshold)
    {
        // Moves the character to the top of the
platform.
        y = other.y - 1;

        // Vertical speed and gravity are reduced to 0
to prevent continuous falling.
        vspeed = 0;
        gravity = 0;

        // Enter the idle state and reset the jump
counter.
        state_id = state_idle;
        entered_new_state = true;
        jump_count = 0;

        // Assigns the current platform
        current_platform = argument0.id;
    }
    // If the character hits a solid collision from
the bottom.
    else if (argument0.solid)
    {
        // Moves the character in reverse of the y
position, similarly to the x-position
        y -= vspeed;

        // Speed is set to 0 to act like they were
stopped.
        vspeed = 0;
    }
}
```

If this script is written correctly, the character should now collide with the two platform types properly. He should be able to walk through the instances of `obj_basic_platform`, creating the illusion that he is walking in front of them. Then, he should be blocked by instances of

`obj_solid_platform`. There is only one problem left to solve. When walking off a platform, Vlad continues to float in the air! A slight change to the **Step** event will resolve this issue quickly.

Updating the Step event

As previously stated, a slight amendment needs to be made to make sure Vlad enters his fall state upon walking off ledges. The highlighted code in the following snippet will do just that:

```
// Jump functionality, which can be performed in
either state except jump.
...
if (state_id != state_jump)
{
    // If above the bottom of the room, the character
falls.
    if (y < global.room_bottom && place_empty(x, y +
1))
    {
        state_id = state_jump;
        grounded = false;
    }
```

So in the **Step** event, after checking whether Vlad is in his jump state or not, his `y` position is checked to see if it is above the bottom of the room, *and* to see if he were moved down one pixel, he would be colliding with anything or not. If both of these are `true`, Vlad will enter his jump state.

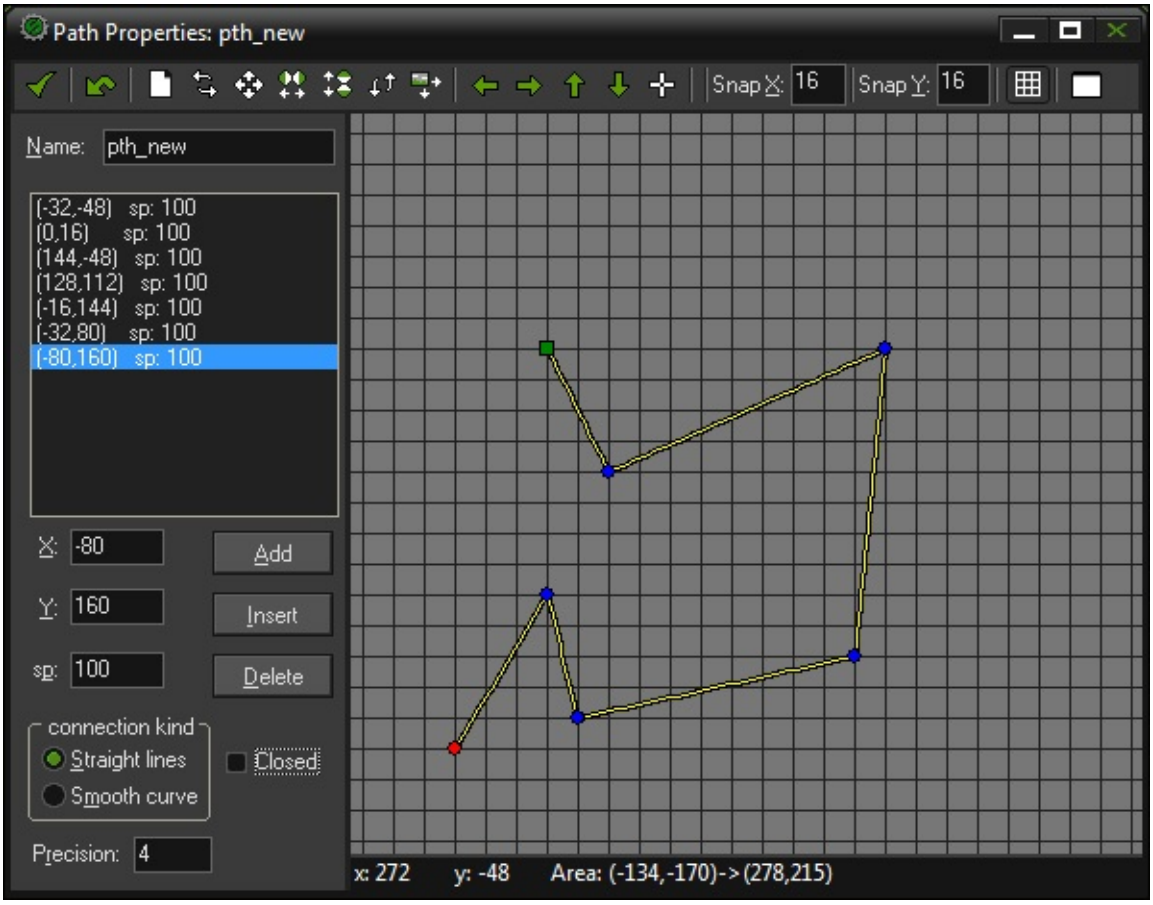
Moving platforms with paths

Stationary platforms, such as the ones created previously, can only do so much. There are a wide variety of platforms that can be created for this type of game, but one popular type involves moving along a path.

Path resources in GameMaker: Studio allow an object to move along a predefined curve that is defined by a set of points. In this section, two simple paths will be created, and a new platform will be constructed that can move along these paths.

Creating path resources

Path resources in GameMaker: Studio can be created by either going to **Resources** | **Create Path** or pressing *Shift* + *Ctrl* + *P*. The **Path Properties** interface with an example path is displayed in the following screenshot:



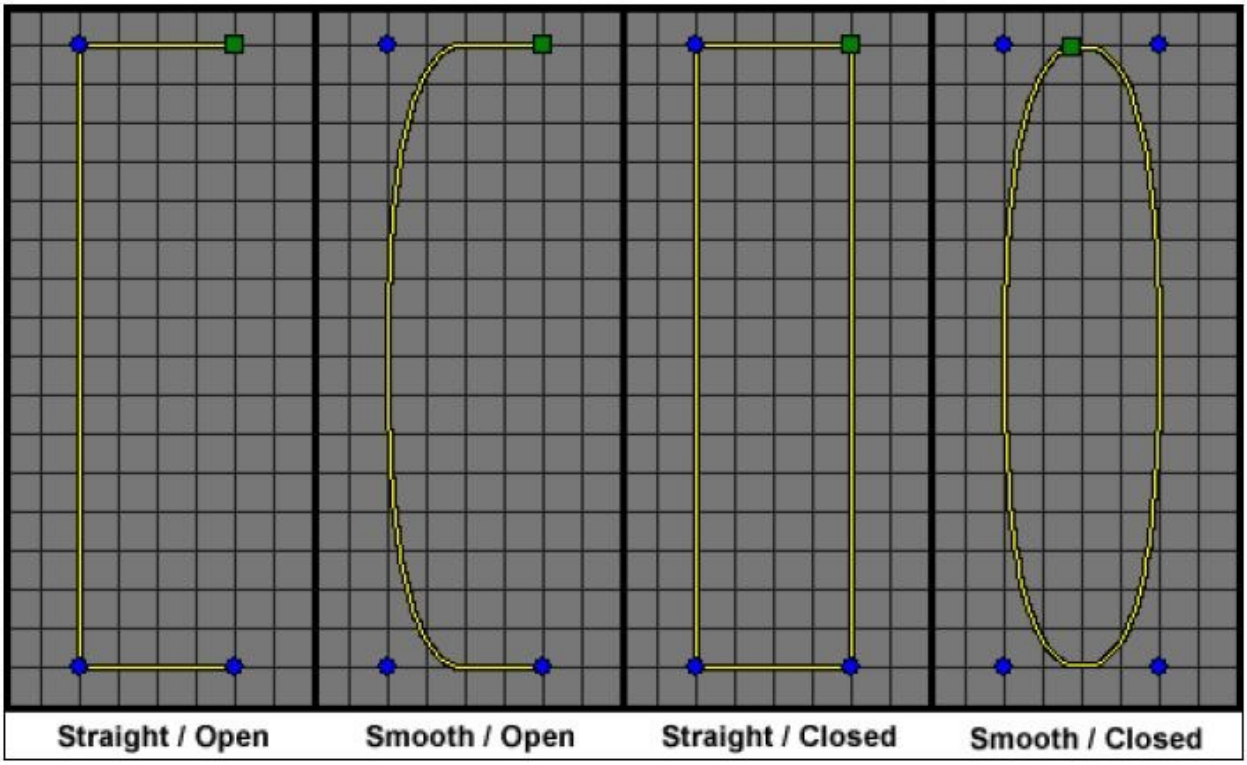
New points on the path can be created by clicking anywhere on the grid. You can also adjust points by clicking-and-dragging them. All points on the path are represented with blue dots. The green rectangle on the curve is the actual starting point on the path itself. When a point is selected, it will change to red.

Below the path's **Name** textfield is a list of points that make up the path. The **X** and **Y** values can be set for the currently selected point. The **sp** term represents the percentage of the path's speed at this point, which is **100** by default. The three buttons to the left of these parameters are as follows:

- **Add**: This button adds a copy of the currently selected point to the end of the path list
- **Insert**: This button inserts a copy of the currently selected point after it in the path list
- **Delete**: This button deletes the currently selected point from the path

list

The **connection kind** section has two choices: **Straight lines** and **Smooth curve**. A path created with straight lines will appear very rigid and angular. A path created with a curve will create a path that smoothly interpolates between the points. The smoothness of these curves is determined by the **Precision** field. **Precision** can be no less than `1` and no greater than `8`. Paths can also be open or closed (this can be decided by checking the **Closed** option). The next screenshot demonstrates the difference between straight or smooth paths and open or closed paths:



Utilizing the path_start function

GameMaker: Studio has a wide variety of functions to manipulate and get information about paths while a game is running. Some of these will be covered later in this chapter, but the ones that are not can be read about in GameMaker: Studio by navigating to **Help** | **Contents...**. Many of these functions are just not important for the current game being created. The most important function concerning paths is `path_start`, which is showcased in the following short code sample:

```
// Moves the current instance along a path.
path_start(path, speed, end_action, absolute);
```

The previous function utilizes the following four arguments:

- `path`: This argument gives the `id` value of the path resource that will be followed.
- `speed`: This is the speed that the current instance will travel at along the path. A negative value will make the instances travel in reverse.
- `end_action`: This argument gives one of the four actions that will occur when the path reaches the end.
- `absolute`: If this argument is set to `true`, the object will travel along the path exactly how it has been defined. If `false`, the path's points will be relative or added to the object's position.

The four available end actions are not defined with built-in constants and can simply be specified using `0`, `1`, `2`, or `3`. The meaning of these four is explained in the following list:

- `0`: Specifying this value ends the movement of the instance on the path.
- `1`: Specifying this value makes the instance continue moving on the path from the starting position, popping to the beginning if not closed.
- `2`: Specifying this value makes the instance continue moving from the current position. For example, if an instance was moving along a simple path with two points—(0,0) to (0,100)—the instance would continue to travel down when reaching the end.
- `3`: Specifying this value reverses the instance's direction on the path by reversing its speed.

It's unfortunate that these do not exist as constants already, but this can easily be remedied. For this game, these four end action values will be represented by the following constants:

- The `path_action_stop` constant, which is set to `0`
- The `path_action_restart` constant, which is set to `1`
- The `path_action_resume` constant, which is set to `2`

- The `path_action_reverse` constant, which is set to `3`

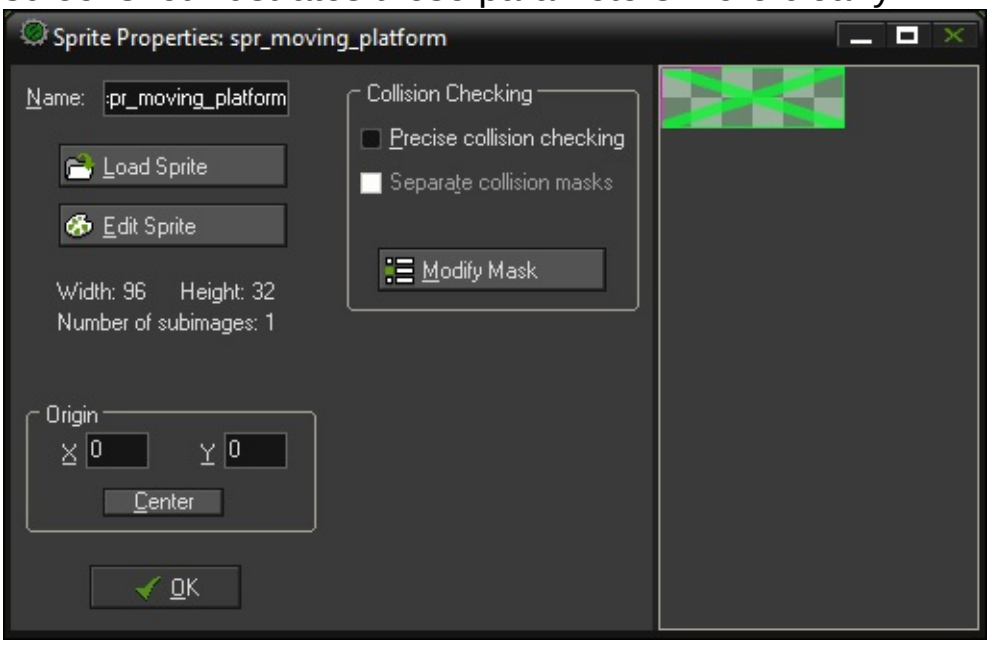
Gathering resources for the path creation

Now that the basics of paths have been covered, two paths will be constructed. The following list includes all of the assets that will be created during the creation of the paths:

- A sprite resource used to represent the objects that will move along the paths
- An object resource that will move along the paths
- Two path resources, `pth_horizontal` and `pth_vertical_ellipse`

The spr_moving_platform sprite

First, a simple sprite resource will be created to represent moving platforms. Unlike `spr_solid_platform` or `spr_horizontal_platform`, this sprite is **96** pixels wide and **32** pixels high. Its origin, like that of the two previously mentioned sprite resources, will also be at the upper-left corner of the sprite. Its mask should not be modified. The next screenshot illustrates these parameters more clearly:

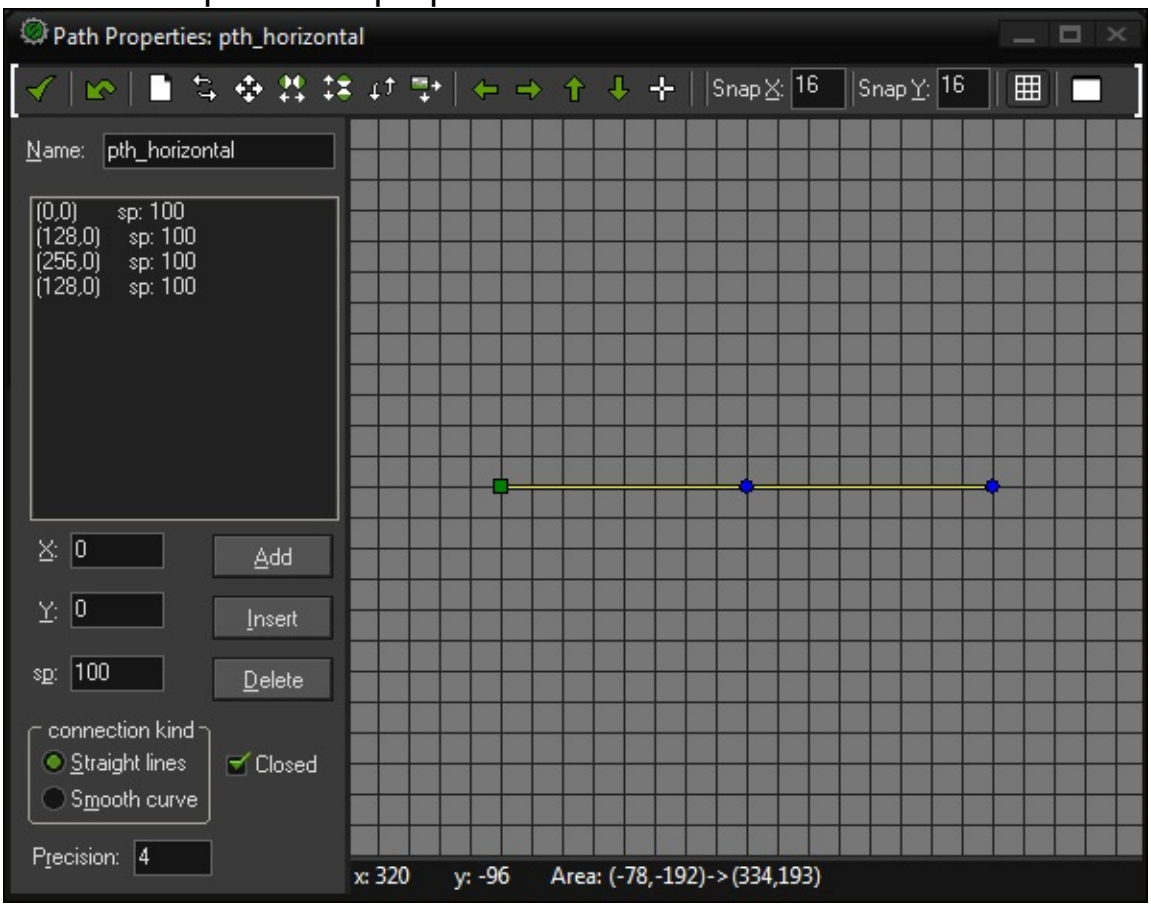


obj_moving_platform

Now that the sprite has been created, an object can be constructed that will be used to represent the moving platforms in the room. The object will use `spr_moving_platform` as its sprite. At present, no events have been created for this object.

pth_horizontal and pth_vertical_ellipse

The first path resource that will be created is named `pth_horizontal`. It is a simple, looping path that will travel back and forth along a horizontal line for **256** pixels. Its properties are illustrated in the next screenshot:

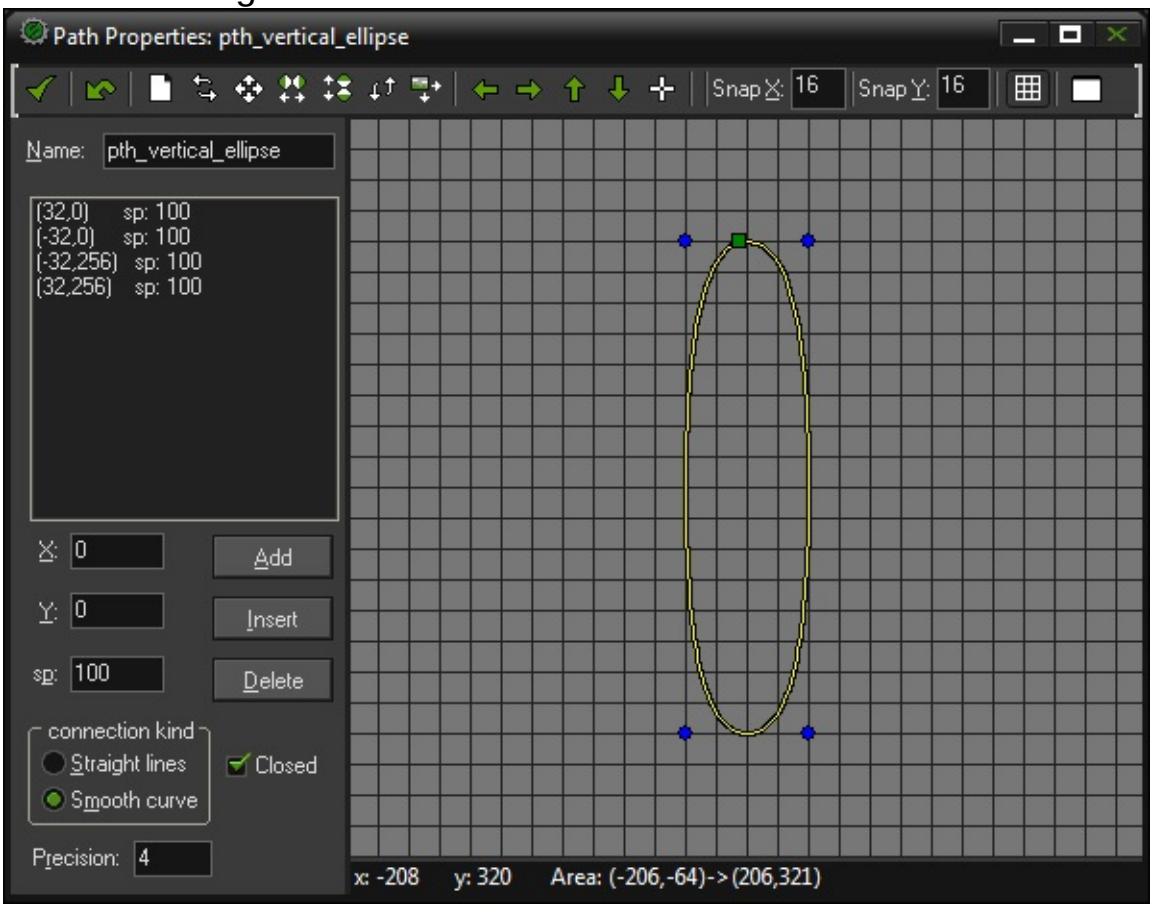


It has the following four points, all that use a speed of `100` percent:

- **(0, 0)**
- **(128, 0)**

- **(256, 0)**
- **(128, 0)**

A second path named `pth_vertical_ellipse` will then be created. This path will also loop, but use a smooth curve. Its parameters are illustrated in the following screenshot:



Like `pth_horizontal`, `pth_vertical_ellipse` also contains the following four points with their speed percentages set to `100`:

- **(32, 0)**
- **(-32, 0)**
- **(-32, 256)**
- **(32, 256)**

Integrating the moving platforms

Before integrating instances of `obj_moving_platform` into the room, several scripts must be modified and events created. These modifications are in the following list:

- The instantiation of `obj_moving_platform` in **Creation Code** of `rm_level1`
- The addition of a new variable in the **Create** event of `obj_vlad`
- The creation of a **Collision** event within `obj_vlad` for colliding with instances of `obj_moving_platform`
- The creation of an **End Step** event for `obj_vlad`
- An update to `scr_test_collision`

Creating instances of obj_moving_platform

As there are multiple paths, instances of `obj_moving_platform` should not be added to the room. Individual object resources could be created for each path, utilizing `path_start` in each one's **Create** event; however, any modifications, such as changing a path's speed, would require the creation of a new object resource. To create object resources that act differently, either unique ones have to be made and their parameters redefined or instances have to be changed individually. To prevent the creation of too many pathed object resources, instances of `obj_moving_platform` will be created within **Creation Code** of `rm_level1`. The positioning of these instances requires some testing and iteration, but these are both important parts of game creation anyway. The code to instantiate moving platforms within this level is as follows:

```
// Creates a vertical moving path
var platform = instance_create(544, 256,
obj_moving_platform);
with (platform)
{
  path_start(pth_vertical_ellipse, 3,
```

```
path_action_restart,false);
}

// Creates a horizontal moving path.
platform = instance_create(16, 96,
obj_moving_platform);
with (platform)
{
  path_start(pth_horizontal, 3, path_action_restart,
false);
}
```

In the previous code, a local variable named `platform` is defined. A new instance of `obj_moving_platform` is then created using the built-in function, `instance_create`. Then, using a `with` statement, the platform is started along a path. The platforms will move along their respective paths at a speed of `3` units using the end action represented by the custom constant `path_action_restart`. Since both paths are closed, this will simply make them loop. Finally, they will move relatively, so no matter where they are created, they will still follow the predefined path.

If the game were to be played now, moving platforms would be visible in the scene, but Vlad would be unable to interact with them. This is because the appropriate **Collision** event for it has still not been created.

Interacting with obj_moving_platform

Now that the instances of `obj_moving_platform` are in the scene, appropriate scripts need to be made so that Vlad can interact with them. First, a **Collision** event for the instances of `obj_moving_platform` should be created within `obj_vlad`. This can be quickly done by right-clicking on either the **Collision** event for `obj_solid_platform` or `obj_basic_platform`, selecting **Duplicate Event**, and then `obj_moving_platform`. This will create a new event with the following code that is already contained within the other two events:

```
/// Collision with instances of obj_moving_platform.

// Runs the collision script.
scr_test_collision(other);
```

Creating a new variable – current_platform

Now, one issue that will occur when Vlad collides with the moving platform is that he won't move with the platform. The platform will slide out from under him. To resolve this, a new variable will be added to the **Create** event of `obj_vlad` as shown in the following code:

```
// The platform that the character is currently on.
current_platform = noone;
```

The variable `current_platform` represents the instance `id` of the platform that Vlad is currently standing on, if any. This variable is used three times as given in the following list:

- It is used in the second **User defined** event used when Vlad jumps
- It is used in the script `scr_test_collision`
- It is used in a new **End Step** event used to move Vlad with the platform

Updating the User Defined 2 event

The update required for the **User defined** event of `obj_vlad`, which is called during the **Step** event when `obj_vlad` is jumping or falling, takes place if `entered_new_state` is `true`. The following code shows this update:

```
// If the state was newly entered.
if (entered_new_state)
{
    // Unassigns the reference to any platform the
character may be standing on.
    current_platform = noone;
```

scr_test_collision and current_platform

Now this is the second time that `current_platform` has been set to `noone`, but when is it actually assigned to something else? The answer to this is within `scr_test_collision`, particularly its second placement check.

These modifications are highlighted in the code portion as follows:

```
// If the the character collides vertically with the
object.
if (place_meeting(x + hspeed, y + vspeed, argument0))
{
    // If the character is moving down and above the
bottom of the platform...
    if (y - vspeed < argument0.y + step_threshold
    && current_platform != argument0.id)
    {
        // Assigns the current platform
        current_platform = argument0.id;
```

So now, when testing if `obj_vlad` is above the collided platform, a test is also performed that compares the current platform's `id` value with the collided instance. If they do not match, which means Vlad is not currently on the platform that was collided with, `current_platform` is set to the `id` value of the object that was collided with, represented by `argument0`. The importance of comparing the `id` values will be explained in the next section with the creation of the **End Step** event in `obj_vlad`.

Using the Step End event

To prevent Vlad from simply sliding off of a moving platform, a **Step End** event is used that will move Vlad with the pathed object. The code for this **Step End** event is as follows:

```
/// Moves Vlad along the current path.

if (current_platform != noone)
{
    // Finds the change on the x-axis of the platform.
    var delta_x = current_platform.x -
current_platform.xprevious;

    // Moves the character along the x-axis with the
platform.
    if (place_free(x + delta_x, y))
    {
        x += delta_x;
    }
```

```
    // Moves the character to the top of the platform
if possible.
    if (place_free(x, current_platform.y - 1))
    {
        y = current_platform.y - 1;
    }
}
```

So first, if `noone` is assigned to `current_platform`, which means that the character is currently not standing on any platform, nothing happens. Because setting a path does not affect `hspeed`, the delta of this current platform is found using its current `x` coordinate and subtracting `xprevious` from it. The `xprevious` variable is a built-in, instanced variable that represents the position of the instance in the previous step. Then, `place_free` is used, and if the sum of the current `x` value and `delta_x` of `obj_vlad` is free of solid collision, Vlad is moved by the delta. Then, if the collision is also free of the solid collision one unit above the platform, Vlad is moved there. These updates will move Vlad above the instances of `obj_moving_platform` and keep him connected to it.

Drawing a path

Because there are various types of paths, a feature utilized in many platformers is to render the moving platform's path. This can be achieved by adding a **Draw** event to `obj_moving_platform` with the following code:

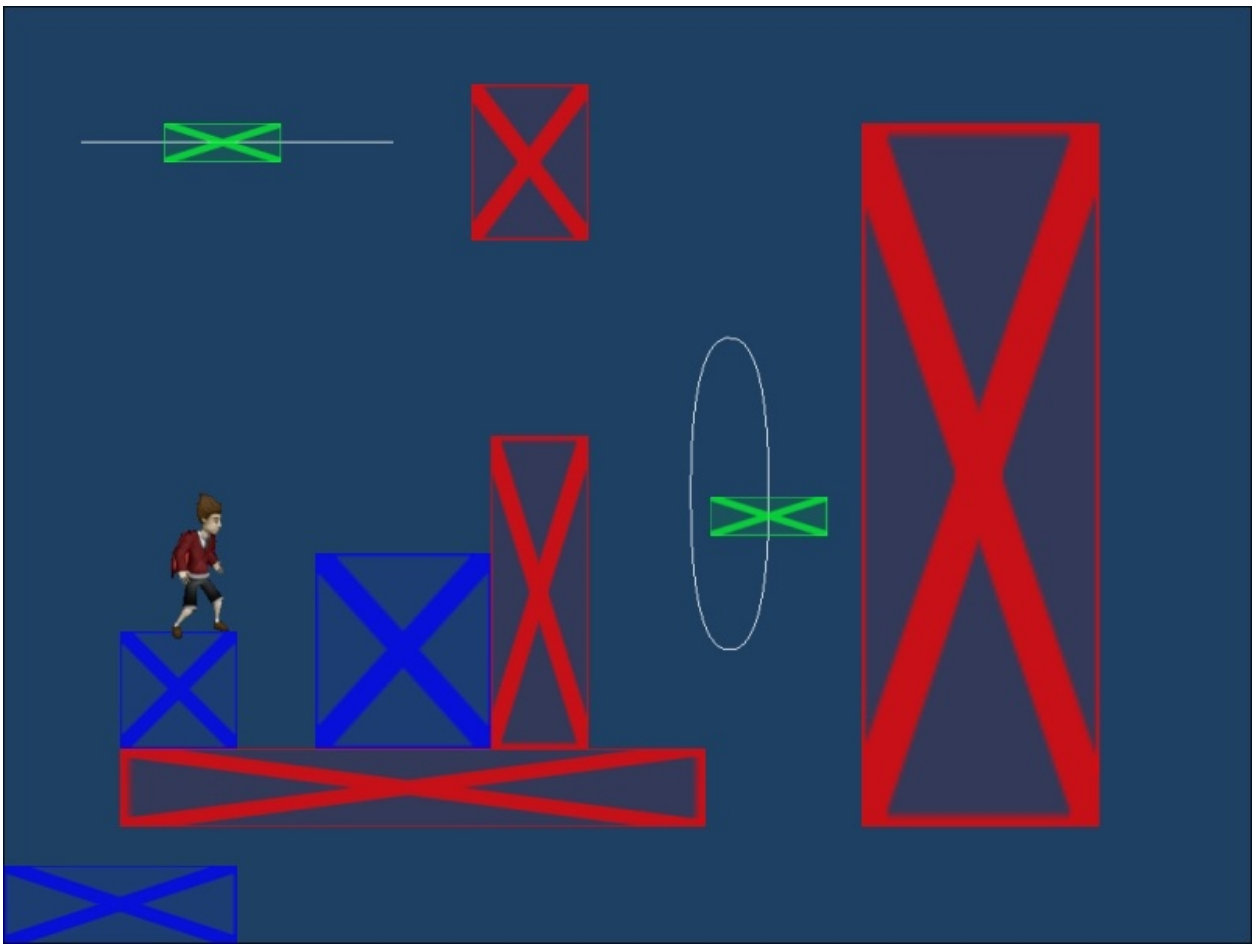
```
// Draws the path if one is present.
if (path_exists(path_index))
{
    // Sets the color.
    draw_set_color(c_white);

    // Draws the path the starting position of the
object.
    draw_path(path_index, xstart + sprite_width  0.5,
ystart + sprite_height  0.5, false);
}

draw_self();
```

First, `path_index`—a built-in variable assigned after executing the function `path_start`—is checked using `path_exists`, a function that checks whether a given path resource is valid or not. If the `path_index` object does exist, then the draw color is set to `white` and the path is drawn using `draw_path`.

The first argument is the index of the path followed by the `x` and `y` positions. The built-in variables `xstart` and `ystart` represent the instance's `x` and `y` positions when the instance was created. The origin of the sprite is determined by individually multiplying the values of `sprite_width` and `sprite_height` by `0.5`; the resulting values are then added to set the starting positions, so the origin is in the center and not the upper left. The final argument represents whether or not the path should be drawn absolutely or relatively as in `path_start`. In this case, it should be drawn relatively, so `false` is provided. After drawing the path, `draw_self` is executed to draw the instance of `obj_moving_platform` on top of this path. If scripted properly, the paths in this scene should be drawn with white lines as shown in the following screenshot (this isn't a necessary step but a nice touch of polish that can be quite useful for players):



Preventing Vlad from leaving

If all the scripts were created properly, Vlad should now be able to move around the room, jumping between platforms, both stationary and moving. There is still one issue that needs to be tackled though: Vlad can leave the room from the sides and disappear. This could be handled by creating two large walls outside of the view, but as the design of the room changes, having to adjust these heights could become cumbersome.

This solution will instead involve the creation of two new global variables, `global.room_left` and `global.room_right`, and updating the **End Step** event of `obj_vlad`.

Defining global.room_left and global.room_right

These two variables will act like `global.room_bottom` to keep the character contained within a defined space. For now, these two variables will match the size of `rm_level1`. They will be defined at the beginning of the **Creation Code** for `rm_level1`. The following code gives the definitions of the bounds of the room:

```
// Defines the bounds of the room
global.room_bottom = 800;
global.room_left = 0;
global.room_right = 1024;
```

Updating the End Step event

Now that the left and right bounds of the room have been defined, the following line of code can be added to the end of the **End Step** event of `obj_vlad`:

```
// Keeps Vlad within the room's bounds.
x = clamp(x, global.room_left, global.room_right);
```

Using the built-in function `clamp`, `x` is set and it is ensured that its value will lie within the range of `global.room_left` and `global.room_right`. As mentioned previously, using this is a much better method of keeping Vlad in the room, as opposed to creating giant instances of `obj_platform_solid`.

Knowing the design ahead of time (when possible)

Before concluding this chapter, it should be noted that collision is a very complex subject; there are many ways to script it that will work. Some approaches are better than others, and there will always be level designs and configurations that can cause issues, such as characters getting stuck, characters falling through platforms, shortcuts that break the flow of the level, and so on. Sometimes, the discovery of these issues won't even happen until the quality assurance phase of a project (if there is one).

To prevent too many surprises, it is important to know what systems will work best for the game that is being made. This can be done by designing the features as early as possible. This, of course, is easier said than done as there are many different scenarios and features that you will encounter when designing a game. Platformers, for instance, have a variety of features such as wall jumping, trampolines, ladders, iciness, and so on. The introduction of even one of these can cause the current implementation of a system to be turned on its head. Sadly, there is no easy way to prevent this problem other than having a strong grasp of the tools being used to make the game.

Summary

In this chapter, collision was discussed as well as placement testing and movement functions. Then, two platforms were built, one of which was solid. Scripts were constructed so the character could interact with these platforms properly when colliding.

Then, path resources were explained, and a moving platform object was created that utilized two paths in the level. Additional scripts were written to account for the changes needed to make Vlad interact with these platforms properly and stick to them as well. A new `draw` function, `draw_path` was introduced to draw the path out for players. Finally, additions were made so that left and right bounds within the room could be defined and used to prevent Vlad from leaving.

Unique levels can now be built for the platformer, but there are some issues. The first is that the room, which is only 1024 x 768 pixels, is beginning to feel a little tight. Even if the size is expanded, Vlad will still be able to leave the room's view. Also, the red, blue, and green crossed rectangles don't really go well with Vlad.

So to fix these issues, views, backgrounds, and tiles will be demonstrated in the next chapter. Views will be used like cameras, to track the character as they move around the environment. This will also allow the room to be larger than 1024 x 768 pixels without affecting the game window. Background resources and tiles will be used to create an environment and atmosphere; this will help give the game some artistic *panache*!

Chapter 8. Setting the Stage – Views, Backgrounds, and Tiles

In the previous chapter, collisions and path resources were utilized to create different platforms for Vlad to jump and run on top of. This chapter will focus on several issues that come up now that the levels have more interactivity.

The first issue is that the current room's size is too small to accommodate any really interesting level designs. Also, if Vlad leaves the bounds of the rooms, he is cropped by the screen. The other issue is that the semi-transparent, square sprite resources created for the platforms are characteristic of what many would call "programmer art", lacking the polish exhibited in many of today's games. The use of a solid-colored background isn't too impressive either.

In this chapter, these issues will be tackled using the following three components of GameMaker:

- Views
- Backgrounds
- Tiles

Views will be used to follow Vlad around and make the room to appear more expansive, allowing for more complex and intriguing level designs. Then, backgrounds and tiles will be used to give this game's environment a stronger sense of setting.

Expanding the room – views

Currently, `rm_level1` is 1024 x 768 pixels in size. All of its contents fit into the screen, but this feels like a rather confined space. Not much can really fit into a room this size, especially when considering that the room isn't even seven full characters high.

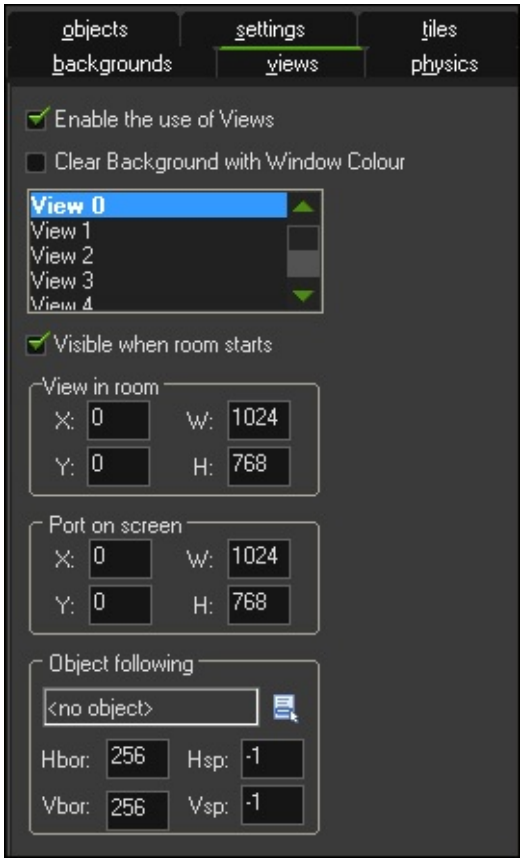
There are several ways in which this room could be expanded. The first is to simply change the dimensions of the room. This, however, presents a problem. When enlarging (or shrinking) a room, the window that displays the game is scaled to fit not only the player's screen but the aspect ratio of the room. This can result in a large window where the player can see the entire level, and is usually not the desired approach.

The best way to achieve this is through the use of **views**. Views are used in GameMaker: Studio to focus on a portion of a room. They can also be used to set the size of the game on the player's screen.

In this section, views will be utilized so that `rm_level1` can be expanded. A new object resource will be created that will follow Vlad and move the view of `rm_level1` so that he is always focused on but will remain within a predefined set of limits.

Setting the view

Views can be edited by clicking on the **views** tab in the **Room Properties** dialog window, as shown in the following screenshot:

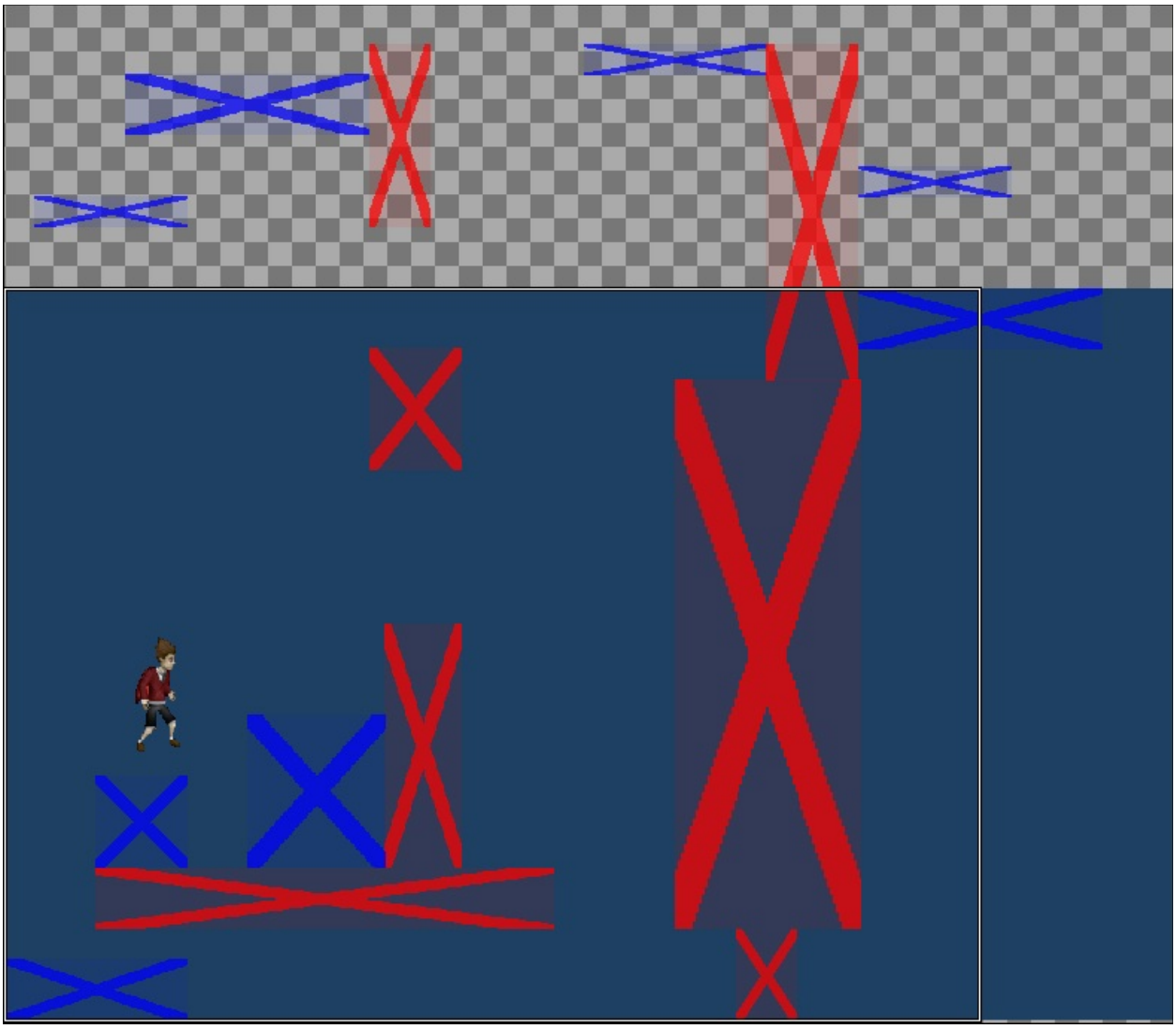


The first checkbox enables the use of views in the room and must be checked so that any view can be active. The **Clear Background with Window Colour** option, when checked, will use a color that can be set with the built-in function `window_set_color`. This is useful if there are multiple views being rendered but not all of them are filling the game window. For the purposes of this game, it can be left unchecked as the current view will fill the entire window.

Then there is a list of eight views (eight being the maximum number of active views allowed in one room at any given time). When checked, the checkbox labeled **Visible when room starts** will allow the game to start using this view immediately. The **View in room** section is a rectangle that describes the area in the room that will be drawn within the view. The **Port on screen** value represents how this view will be translated to the final game window. This can be used to make the game scale to different resolutions, but can also cause stretching. For example, if the game is ported to 1024 x 768 pixels on screen but the view is 1280 x 720 pixels, there will be stretching because these are different **aspect ratios**, 4:3

and 16:9. **Object following** allows a type of object resource to be selected for this view to follow. **Hbor** is the horizontal border, meaning that when this object reaches the edge of the view by this much, the view will move horizontally. **Vbor** does the same, but vertically. **Hsp** represents the horizontal speed at which the view will move when the object crosses the border; **Vsp** represents the vertical speed. Setting either of these to `-1`, which is the default, will make the views move automatically.

Now that views have been introduced, the room can be expanded. In the following screenshot, the room has been expanded to 1280 pixels wide. The view at 1024 x 768 pixels is highlighted by a white, outlined rectangle in the lower-left part of the screen. This room's new design also includes platforms above and to the right of the room's original bounds.



Now, **View 0** can be set to follow Vlad around the room. In this game, the view will be set to the same size that the room was originally and the port will match the view to prevent any scaling. Then, the object followed will be set to `obj_vlad` with the **Hbor** and **Vbor** values set to `256` and **Hsp** and **Vsp** set to `-1`.

If the game were to be played now, the view should follow Vlad when he gets less than 256 pixels away from the edge of the screen horizontally. Unfortunately, the view will not leave the bounds of the room, and since there are platforms above it, Vlad will leave the view when this occurs. One way to fix this could be to increase the height of the room, but this would require a lot of work. All of the current objects would have to be moved down. The same applies if the room were to be expanded

horizontally to include more space on the left. It should never be assumed that one will know the exact size of a level before they start designing and building, and adjustments like this can become rather cumbersome.

To help this process, a new object resource will be created that will adjust the current view to follow Vlad; at the same time, it will use predefined and adjustable limits so that Vlad can travel outside of the room bounds if needed.

Note

For the remainder of this chapter, **Object following** should be unassigned.

Adjusting view parameters

Before creating the object resource that will act as our camera, the way in which views' different parameters are accessed should be covered. These parameters are adjusted through the use of several arrays, most of which are defined in the following code sample:

```
/// View parameters.  All arrays range from 0 to 7.




// If true, allows the room to use views.

view_enabled = true;




// If true, the current view will be visible.

view_visible[0] = true;




// The x position of the view at the supplied index
(in this case View 0).

view_xview[0] = 0;
```

```
// The y position of the view at the supplied index.

view_yview[0] = 0;



// The width of the view at the supplied index.

view_wview[0] = 1024;



// The height of the view at the supplied index.

view_hview[0] = 768;



// The x position of the view ported to the game
window.

view_xport[0] = 0;
```

```
// The y position of the view ported to the game
window.


view_yport[0] = 0;




// The width of the view ported to the game window.


view_wport[0] = 1024;




// The height of the view ported to the game window.


view_hport[0] = 768;




// The object instance that will be followed.


view_object[0] = instance_find(obj_vlad, 0);


// The horizontal border of the view.


view_hborder[0] = 128;
```

```
// The horizontal speed of the view.

view_hspeed[0] = -1;



// The vertical border of the view.

view_vborder[0] = 128;



// The vertical speed of the view

view_vspeed[0] = -1;



// In degrees, allows views to be rotated.

view_angle[0] = 0;



// Read-only variable that returns the current view
being rendered and is only used by Draw Events.
```

```
if (view_current == 0)
                            {
    draw_text(0, 0, "View 0");
}
```

Preparing the game for obj_camera

Now that the basic view parameter variables have been covered, the camera object can be made. Only one new object resource needs to be created; this object resource will be named `obj_camera`. It will have a **Create** event and an **End Step** event. The **Create** event will mostly define variables, while the **End Step** event will be used to actually move the camera.

The Create event

As with many of the object resources previously created in this book, the **Create** event of `obj_camera` will define a variety of variables. The following script will be executed by the **Create** event:

```
/// Initializes variables for the camera.

// The index of the view that will be manipulated.
view_index = 0;

// The left most limit that the camera can move.
limit_left = 0;

// The right most limit that the camera can move.
limit_right = room_width;

// The top most limit that the camera can move.
limit_top = 0;

// The bottom most limit that the camera can move.
limit_bottom = room_height;

// The horizontal offset of the view, half of the
view's width, to focus on the character.
offset_factor_x = 0.5;

// The vertical offset of the view, about a two thirds
of the vertical view space.
```

```
offset_factor_y = 0.667;

// The rate at which the camera moves towards the
target.
lerp_rate = 0.1

// Tries to assign the target by looking for an
instance of the character.
target = noone;
```

Again, views' parameters are accessed in scripts through the use of arrays, so it's a good idea to define the index of the view that will be manipulated in the **End Step** event. Then, `limit_left` through `limit_bottom` are defined, representing the range in which the camera will be able to move, currently set to match the room's size. The `offset_factor_x` and `offset_factor_y` values are percentages of the view that the character will be focused on. The `offset_factor_x` variable being set to `0.5` will center the character horizontally, and `offset_factor_y` will place the character two-thirds from the top of the view. The `lerp_rate` value represents the speed at which the view will reach its target destination. The closer to `0`, the longer it will take to reach its destination, and the closer to `1`, the faster it will reach its destination. This value should fall between `0` and `1` as values outside this range can cause undesirable results, either overshooting or moving away from the target. Finally, `target` represents the instance's `id` value that this camera will follow, but is set to `noone` by default.

The End Step event

Because any target objects could possibly change their position during other **Step** events, using an **End Step** event to move the view is the best choice in the hope that its target, if there is one, will no longer move. The following is the code for the **End Step** event of `obj_camera`:

```
/// End Step function for the camera.

// Moves the camera towards the target if one exists.
if (target != noone)
{
```

```
    var offset_x = -view_wview[view_index]
offset_factor_x;
    var offset_y = -view_hview[view_index]
offset_factor_y;

    x = lerp(x, target.x + offset_x, lerp_rate);
    y = lerp(y, target.y + offset_y, lerp_rate);
}

// Clamps the view between the left and right bounds
of the camera.
x = clamp(x, limit_left, limit_right -
view_wview[view_index]);
y = clamp(y, limit_top, limit_bottom -
view_hview[view_index]);

// Moves the view to the x and y of the camera.
view_xview[view_index] = x;
view_yview[view_index] = y;
```

The first section checks to see if `target` is assigned to an object. If so, two local variables are created, `offset_x` and `offset_y`. These are created by taking the negative size of the room and multiplying it by the previously defined percentages and will be used to determine the position of the view. Then, these values are added to the the target's position and linearly interpolated to the current `x` and `y` values of `obj_camera`. Using the negative width and height of the view actually puts the view to the left of and above the target. Moving the view directly to the target would lock the target to the upper-left of the room.

Then, the camera is clamped between the previously defined limits. Since the view's width and height could be changed, subtracting these values from `limit_right` and `limit_bottom`, respectively, gives more flexibility. Finally, the view's `x` and `y` values are set to the `x` and `y` values of the camera.

Adding the camera with Creation Code

Now that `obj_camera` has been created, it can be added to `rm_level1`. By default, the limits are set to the size of the room, so this doesn't solve the earlier problem; the character will still be cut off when he reaches the top

of the screen. So, instead of adding an instance of `obj_camera` to the room, it will be created with the following addition to the **Creation Code** page of `rm_level1`:

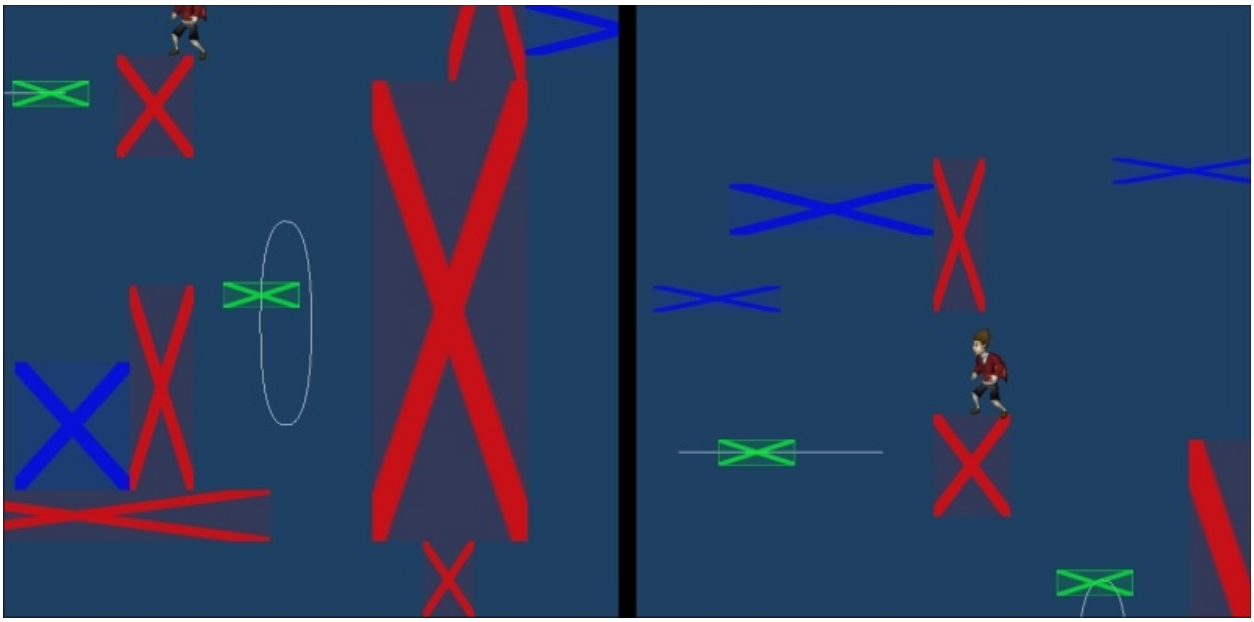
```
// Creates the camera.
var camera = instance_create(0,0, obj_camera);

// Sets the top limit to be above the room.
camera.limit_top = -512;

// Finds the first instance of obj_vlad and assigns it
to this camera.
camera.target = instance_find(obj_vlad, 0);
```

In the previous code, a new instance of `obj_camera` is created. The `limit_top` value is then set to `-512`, which will make the view go 512 pixels above the room. Finally, the first instance of `obj_vlad` will be found using the built-in function `instance_find`, whose first argument is the object resource's index and second variable represents the number of the instance to find based on the order in which they were created. If there were two instances of `obj_vlad`, using `0` would find the first instance and using `1` would find the second.

With the introduction of this creation code, Vlad should be able to go higher up in the level and not get cut off on screen despite the fact that the room was not built to accommodate this new height. In the following screenshot, the left-hand side section represents Vlad in the room before adjusting `limit_top`, and the right-hand side section represents what it should look like now:



Setting the environment – backgrounds

Vlad is currently being followed by a camera of some sort, but he still doesn't really fit into an environment. One reason for this is the lack of a background; in this section, a **background resource** will be created that will be moved by `obj_camera` and hopefully will give the game a better sense of setting.

Introducing background variables

Before jumping to the creation of a background resource or related scripts, variables associated with backgrounds will be discussed. Backgrounds' parameters are very similar to views', as their parameters are accessed through an array. The following code sample showcases these different parameters:

```
// The Resource id of the specified background.
background_index[0] = bg_main

// Determines if the background is visible or not.
background_visible[0] = true;

// Sets the alpha of the background, similar to
image_alpha.
background_alpha[0] = 1.0;

// Sets the blend color of the background, similar to
image_blend.
background_blend[0] = c_white;

// The horizontal position of the background
background_x[0] = 0;

// The vertical position of the background.
background_y[0] = 0;

// Is the background being used as a foreground?
background_foreground[0] = false;
```

```
// The horizontal speed of the background.
background_hspeed[0] = 0;

// The vertical speed of the background.
background_vspeed[0] = 0;

// Is the background tiled horizontally?
background_htiled[0] = false;

// Is the background tiled vertically?
background_vtiled[0] = false;

// The width of the background.  Cannot be set.
if (background_width[0] < 1024)
{
  return "Background is not wide enough.";
}

// The height of the background.  Cannot be set.
if (background_height[0] < 768)
{
  return "Background is not tall enough.";
}

// The horizontal scale of the background.
background_xscale[0] = 1.0;

// The vertical scale of the background.
background_yscale[0] = 1.0;

// If true, a specified background color will be drawn
if no
// background is currently present.  Using false with
no
// background will result in graphic artifacts.
background_showcolor = true;

// The background color drawn if background_showcolor
is true.
background_color = c_blue;
```

Creating background resources

Background resources are similar to sprite resources in that both are

defined with an image. Unlike sprite resources, each background is only defined by a singular image and not a series.

The **Background Properties** dialog window, displayed in the next screenshot, is rather simple:



The **Load Background** button will bring up a file manager so that an image can be selected. The **Edit Background** button will open GameMaker: Studio's image editor so that the background can be edited.

> ## Note
>
> The checkbox labeled **Use as a tile set** will be discussed later in this chapter.

Preparing the game for background resources

Before explaining how to utilize a background, a background resource must be created. Then, a minor update will be made to the **Create** event

of `obj_camera` and a more substantial update will be made to its **End Step** event. Also, a new script resource will be written, named `scr_inverse_lerp`, which will aid in moving the background during the **End Step** event of `obj_camera`.

Building an atmosphere with bg_main

As mentioned previously, Vlad is a vampire, so using a background to help sell this is important. The next screenshot is what will be used for the new background resource, `bg_main`, which is of full moon shining over a lake with trees and isolated houses in silhouette and will be used for the background in `rm_level1`.



This background image was created by Indie Squid (http://indiesquid.com)

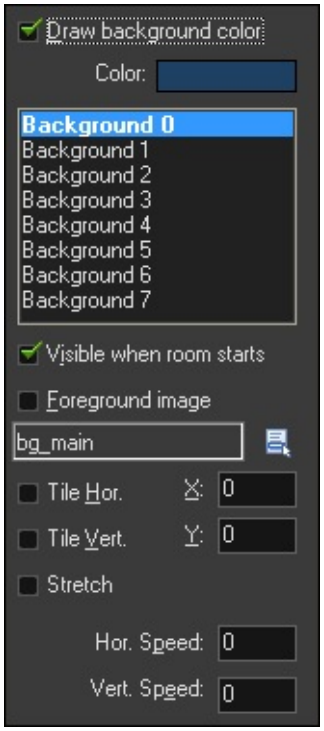
The fact that it is a night environment with subtle hints of spookiness seems like an appropriate environment for the character, and even this simple background can give a sense of story to the game and make the players ask questions. *Is the house Vlad's? Is it an enemy's? Is Vlad*

trying to make it home before the sun rises? Regardless of the questions asked, they are all better than the ones asked when the background is just a solid color. Now this image is a little larger than the port size defined earlier. Initially, it will be cut off, but again, code will be written later to help remedy this.

Utilizing the background

Now that `bg_main` has been created, it can be used within the room `rm_level1`.

In the **Room Properties** dialog window, there is a **backgrounds** tab. This tab allows a background to be defined. Similar to views, eight backgrounds can be defined. The next screenshot shows the parameters for using `bg_main` as **Background 0**:



Similar to views, **Visible when room starts** will make use of the background upon entering the room. **Foreground image** will bring the background in front of all objects in the room. If this were to be done now, everything would be covered, so it will be unchecked for `bg_main`. Below

this checkbox, `bg_main` can be defined by selecting it from a list of existing background resources.

Tile Hor. and **Tile Vert.** will repeat the background image vertically and horizontally to fill the room. The **X** and **Y** parameters to the right of these will offset the background. The **Stretch** checkbox will stretch the background to fit within the room's bounds. Finally, **Hor. Speed** and **Vert. Speed** will move the background by the value set in the respective direction. The `bg_main` object will be manipulated through a script, so for now, these parameters can either be left unchecked or set with the value `0`.

Setting a background index with bg_index

Now that the basic parameters for backgrounds have been discussed, scripts can be written to move the background with the camera.

If the game were to be played now, `bg_main` would remain static. Eventually, the background would only be displayed part way, revealing the solid color that previously resided there. The goal would be that any room, as long as it is larger than the background image, should be visible to the camera at all times.

The first script update is for the initialization of a simple variable, `bg_index`. This variable will be the index of the background moved by the camera. This variable can be added to the script at the bottom of the **Create** event of `obj_camera` as shown in the following code:

```
// The index of the background that will be accessed.
bg_index = 0;
```

Since there is only one background, `bg_main`, being utilized this will be set to `0`.

Scripting scr_inverse_lerp

Before updating the **End Step** event of `obj_camera`, a new script resource will be made. By default, GameMaker: Studio has the ability to obtain a number between two values when provided with a percentage using the `lerp` function. Unfortunately, it does not provide the opposite: the percentage when provided a range and a number associated within that range. The following script does that, taking some cautionary measures to prevent errors:

```
// Gets the percent of a value between a minimum and
maximum value.

/* arguments
argument0 -- the minimum value
argument1 -- the maximum value
argument2 -- the value being tested.
*/

var minimum, maximum, test_value;
minimum = argument0;
maximum = argument1;
test_value = argument2;

// To avoid errors zero division errors, if minimum
and maximum are the same, 0 is returned.
if (minimum == maximum)
{
    return 0;
}

// Returns the difference of the value and the minimum
divided by the difference of the maximum and the
minimum.
return (test_value - minimum) / (maximum – minimum);
```

In the previous script, the three arguments are first assigned to local variables for clarity. Then, `minimum` and `maximum` are compared. If they are identical, `0` is returned. This is because of the next section, in which the difference of `test_value` and `minimum` is divided by the difference of `maximum` and `minimum`. If `maximum` and `minimum` are equal, then their difference will be `0`. Trying to divide by `0` will result in a runtime error, so to prevent this, the values of `minimum` and `maximum` are checked beforehand.

Moving the background in the End Step event of obj_camera

Instances of `obj_camera` are moved within the **End Step** event, so the background will also be moved in this event after the camera's movement. This *could* easily be done by using the following code:

```
// Aligns the background to the view.
background_x[bg_index] = x;
background_y[bg_index] = y;
```

The one issue with the previous code is that it would make the background stick to the camera, acting like a static backdrop. Also, since this image is larger than the current view, much of it is being unused. Instead, this background, while still moving with the camera, will do so at a rate that will make it move a bit out of sync, making it appear as if it is far away.

The idea of this update is to set the position of the background based on where the camera currently is between its limits. Doing this will move the background at a different rate than the camera, fill the screen at all times, and allow the player to see all of the background. To do this, several pairs of background and camera positioning values must be defined, as mentioned in the following list:

- The minimum `x` and `y` positions at which the background can be placed
- The maximum `x` and `y` positions at which the background can be placed
- The minimum `x` and `y` positions up to which the camera can move
- The maximum `x` and `y` positions up to which the camera can move
- The percentage of where the camera falls between its respective minimum and maximum values

The following script should be added to the **End Step** event of `obj_camera`; it demonstrates how these values are derived and later used:

```
/// Adjusts the background.

// Gets the minimum and maximum ranges that the
background can move.
var bg_x_min = limit_left;
var bg_y_min = limit_top;

var bg_x_max = max(limit_left, limit_right -
background_width[bg_index]);
var bg_y_max = max(limit_top, limit_bottom -
background_height[bg_index]);

// Gets the minimum and maximum ranges of where the
camera will move.
var cam_x_min = limit_left;
var cam_y_min = limit_top;

var cam_x_max = max(limit_left, limit_right -
view_wview[view_index]);
var cam_y_max = max(limit_top, limit_bottom -
view_hview[view_index]);

// Gets the percent of the camera between its left and
right limits
var x_percent = scr_inverse_lerp(cam_x_min, cam_x_max,
x);
var y_percent = scr_inverse_lerp(cam_y_min, cam_y_max,
y);

// Sets the x and y of the background based on percent
of the camera within its range.
background_x[bg_index] = lerp(bg_x_min, bg_x_max,
x_percent);
background_y[bg_index] = lerp(bg_y_min, bg_y_max,
y_percent);
```

In the previous code, the values listed earlier are derived using the camera's four limit values: `limit_left`, `limit_right`, `limit_top`, and `limit_bottom`. When deriving the maximum values, the built-in function `max` is used to prevent the maximum value from being less than the minimum value by returning the minimum value instead, as shown in the following code:

```
var bg_x_max = max(limit_left, limit_right -
```

```
background_width[bg_index]);
var bg_y_max = max(limit_top, limit_bottom –
background_height[bg_index]);
```

The previous code is repeated when finding the maximum position vertical to which the camera can move. Then, the script mentioned earlier, `scr_inverse_lerp`, is used. This script obtains the percentage value between the position of `x` and `y` of `obj_camera` within the camera minimums and maximums. Then, these percentages are used to place the background between its minimum and maximum values, using the built-in function `lerp`.

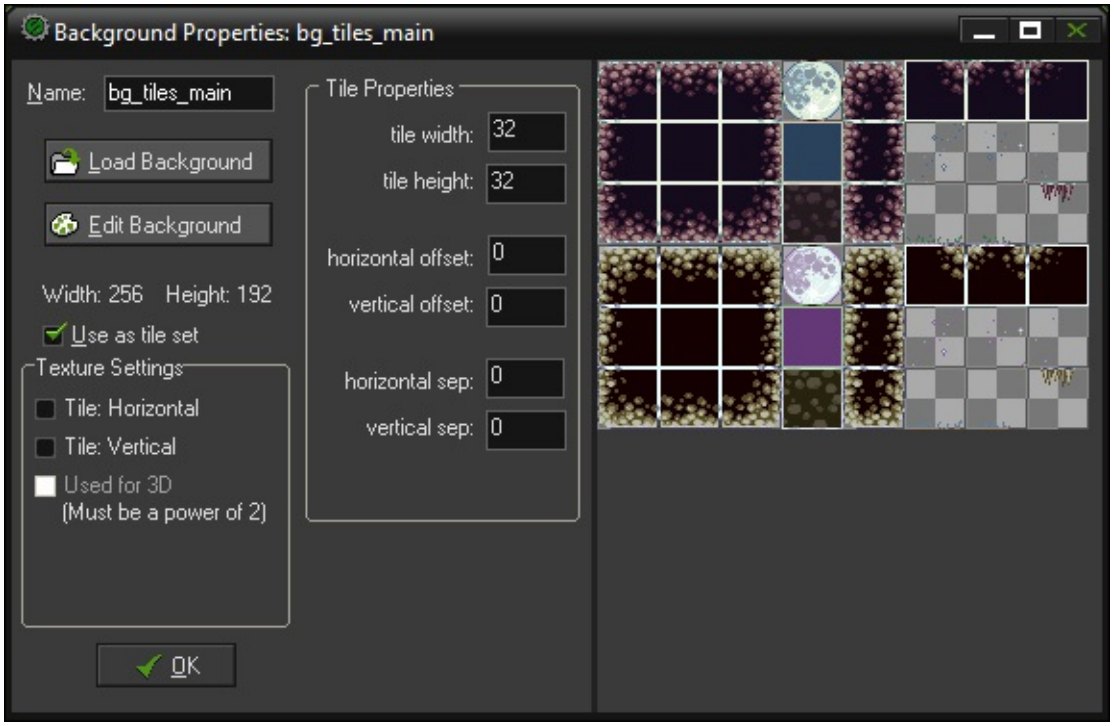
If the scripts were implemented correctly, the background should now move with the camera but at a different rate than before, making it appear to be farther away from the faster-moving objects in the foreground. This is a simplified form of a visual effect often known as **parallax**.

Introducing tiles

In the previous section, backgrounds were implemented to help flesh out the environment; however, one important aspect of the game still appears to be using very primitive art: the platforms. One quick solution for this could be to replace the sprite resources used by the platforms, but because the platforms are being scaled to fulfill a variety of sizes, the sprites would be stretched, appearing extremely jagged or pixelated. To remedy this issue, **tiles**, which are a component in GameMaker: Studio just as backgrounds are, will be utilized.

Creating tiles

A set of tiles can be created in the **Background Properties** dialog window. Upon checking the **Use as a tile set** checkbox, additional parameters as well as a grid over the currently displayed background will appear, as shown in the following screenshot:



The parameters are used to define the tiles as follows:

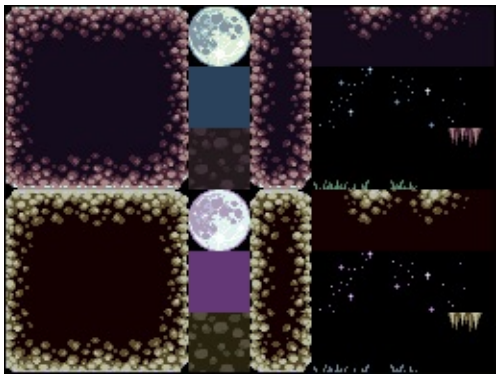
- **tile width**: This parameter gives the horizontal size of the tile
- **tile height**: This parameter gives the vertical size of the tile
- **horizontal offset**: This parameter gives the amount of space from the left of the image to start tiling
- **vertical offset**: This parameter gives the amount of space from the top of the image to start creating tiles
- **horizontal sep**: This parameter gives the spacing between tiles on the *x* axis
- **vertical sep**: This parameter gives the spacing between tiles on the *y* axis

Building resources for tiles

Now that the creation of tiles has been discussed, a set of tiles will be created for use in this game. These tiles will be known as `bg_tiles_main`.

Tiling with bg_tiles_main

So far, the environment being defined is an outdoor one. To establish this further, the tiles that will be created will have an earthy look to them. These tiles are pictured in the following screenshot:

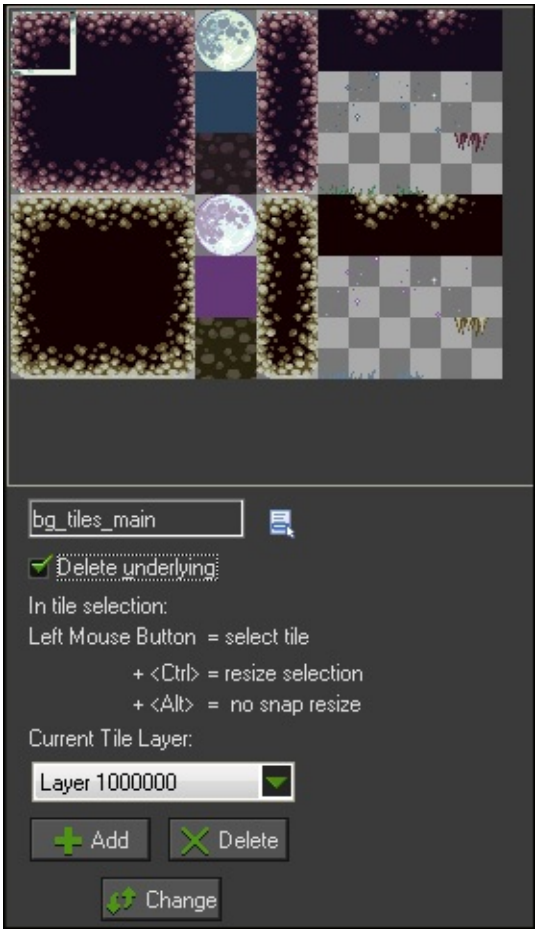


This tilesheet was created by Lanea Zimmerman

Each tile is 32 x 32 pixels with no horizontal or vertical separation or offset. The top half of the screenshot is the same as the bottom half but with a slightly more purple tone. The reasoning for this will be explained later.

Applying tiles

Now that a set of tiles has actually been created, they can be placed into the scene. This is done while the **tile** tab is active in the room properties dialog window, as shown in the following screenshot:



First, a background resource can be chosen that will be used to place tiles in the scene; in this case, `bg_tiles_main`. Once a background is chosen, a tile from the grid defined in **Background Properties** can be selected. Once selected, tiles can be "painted" into the scene by clicking anywhere in the room.

The **layer** in which a tile is placed can also be defined. The higher the layer's depth, the further back it will appear in the background. Also, an entire layer can be deleted, which will remove all tiles on that layer from the room.

Now, in `rm_level1`, tiles could be placed for every platform, but this could become rather tedious as some platforms are rather large. It can also be troublesome if the design of the level needs to change and a large number of tiles has to be removed. In the next section, a script will be created so that all of the necessary tiles are placed upon the creation of instances of `obj_platform_basic` and `obj_platform_solid`. Then, instances of `obj_platform_moving` will utilize the **Draw** event to draw tiles.

Utilizing tile_add and other tile functions

When creating tiles dynamically in the **Create** events of the different platforms, the most important function that will be used is `tile_add`. This function returns an `id` value that can be used by other functions to manipulate the newly created tile. The `tile_add` function is shown in the following code sample:

```
// Adds a tile to the scene, returning its id.
var tile_id = tile_add(bg_tiles_main, 0, 32, 64, 64,
192, 24, 1000000);
```

The arguments for this function are as follows:

- `argument0`: This argument gives the index of the background resource to get the tile from
- `argument1`: This argument gives the `x` position in the background resource to get the tile
- `argument2`: This argument gives the `y` position in the background resource to get the tile
- `argument3`: This argument gives the width of the tile to be created
- `argument4`: This argument gives the height of the tile to be created
- `argument5`: This argument gives the `x` position in the room where the tile is to be created
- `argument6`: This argument gives the `y` position in the room where the tile is to be created
- `argument7`: This argument gives the depth at which the tile is to be created

Note

Tiles can be manipulated in a similar manner to objects using a variety of different functions prefaced by `tile_set`; however, these will not be gone over in this text in detail. For more details on these, GameMaker: Studio's help contents can be referred to.

Placing tiles with scripts – scr_define_tiles

Now that `tile_add` has been discussed, the way in which tiles will be placed in the **Create** events of `obj_solid_platform` and `obj_basic_platform` can be explained. Since there are two types of platforms, different tiles sets should be used to help the player differentiate between the two types. To achieve this, `bg_tiles_main` contains duplicated tiles that only differentiate slightly in hue in the same texture.

The idea now is that platforms will be **nine-sliced**, or divided into nine sections. The corners of the platform will be unique, but the top and bottom will repeat horizontally, the left and right will repeat vertically, and the middle will be duplicated as often as it is needed to fill the given space. Using this idea, a script resource named `scr_define_tiles` will be created and used in the **Create** events of the two static platforms to define different tiles.

The following script takes the provided arguments to build a series of tiles that will recreate the platforms. The first section, which is next, tries to determine the size and scale of the `x` and `y` tiles:

```
/// Lays out tiles based on the shape and size of the
given object.

/* arguments
argument0 -- the width of the object.
argument1 -- the height of the object.
```

```
argument2 -- the xspacing between tiles
argument3 -- the yspacing between tiles
argument4 -- the y offset of the tile set being drawn,
separating the two sets.
argument5 -- the layer of the tiles.
/

var obj_w = argument0;
var obj_h = argument1;
var tile_xspace = argument2;
var tile_yspace = argument3;
var tile_yoffset = argument4;
var tile_layer = argument5;

var x_size, y_size, x_scale, y_scale;

// If the width of the object is less than two
tiles...
if (obj_w < tile_xspace  2)
{
    // Size is half of the width.
    x_size = 0.5 * obj_w;
}
else
{
    // The size of each segment is the size of the
tile, minus the remainder of the width
    x_size = tile_xspace - (obj_w mod tile_xspace);
}

// The scale of each tile is the size divided by the
original size.
x_scale = x_size / tile_xspace;
```

Firstly, four local variables are defined. The variables `x_size` and `y_size` are the size of each tile, while `x_scale` and `y_scale` are the scales used to achieve these sizes. Then, if the width of the object is less than two tiles, which would be the left and right tiles combined, the size is determined to be half of the respective argument; in this case, `obj_w`. The respective scale is then determined by using the recently assigned size and dividing it by the size of the tile.

After determining the size and scale needed to create the tiles, the tiles are added to the scene through the following iterative loop:

are added to the scene through the following iterative loop:

```
// Determines the y size and scales similarly to the
above.
if (obj_h < tile_yspace * 2)
{
    y_size = 0.5  obj_h;
}
else
{
    y_size = tile_yspace - (obj_h mod tile_yspace);
}

y_scale = y_size  tile_yspace;

/ Iterates along the width and height of the object by
the size of the tile.
for (var i = 0; i < obj_w; i += x_size)
{
    for (var j = 0; j < obj_h; j += y_size)
    {
        var tile_x, tile_y;

        // Determines the x coordinate of the tiles
within the tile texture.
        if (i == 0)
            tile_x = 0;
        else if (i + x_size < obj_w)
            tile_x = tile_xspace;
        else
            tile_x = tile_xspace  2;

        // Determines the y coordinate of the tiles
within the tile texture.
        if (j == 0)
            tile_y = 0;
        else if (j + y_size < obj_h)
            tile_y = tile_yspace;
        else
            tile_y = tile_yspace * 2;

        // Adds a new tile to the scene based on the
parameters derived above.
        var new_tile = tile_add(bg_tiles_main, tile_x,
tile_y + tile_yoffset, tile_xspace, tile_yspace, i +
x, j + y, tile_layer);
```

```
            // Scales the newly created tile.
            tile_set_scale(new_tile, x_scale, y_scale);
        }
    }
```

This explanation will focus on `i`, which relates to the horizontal values, while `j` relates to the vertical placement. In the `for` statement, these variables are not incrementing by one, but instead are increased by the sizes defined earlier, and the `for` statements are continued as long as they remain smaller than the respective sprite sizes. Then, the coordinate within the background texture, `tile_x`, is determined. If `i` is equal to `0`, meaning it is the first tile in the series, `tile_x` is set to `0`, using the leftmost portion of texture. However, if the sum of `i` and `x_size` is less than the width of the platform, `obj_w`, the middle tile will be used by setting `tile_x` to `tile_xspace`; otherwise, `tile_x` is set to `tile_xspace` multiplied by `2`, getting the right side. This is repeated for `j` and the `y` coordinates. Doing this will ensure that all nine different sections of the texture are used.

Then, the tile is created by using `tile_add`. The `y` position of the tile is offset by `tile_yoffset`, which will allow a different type of tile to be used with the same background resource. Finally, the newly created tile is scaled using `tile_set_scale`.

This script is a bit lengthy, so here is the script repeated in its entirety:

```
/// Lays out tiles based on the shape and size of the
given object.

/* arguments
argument0 -- the width of the object.
argument1 -- the heigth of the object.
argument2 -- the xspacing between tiles
argument3 -- the yspacing between tiles
argument4 -- the y offset of the tile set being drawn,
separating the two sets.
argument5 -- the layer of the tiles.
/

var obj_w = argument0;
var obj_h = argument1;
```

```
var tile_xspace = argument2;
var tile_yspace = argument3;
var tile_yoffset = argument4;
var tile_layer = argument5;

var x_size, y_size, x_scale, y_scale;

// If the width of the object is less than two
tiles...
if (obj_w < tile_xspace  2)
{
    // Size is half of the width.
    x_size = 0.5  obj_w;
}
else
{
    // The size of each segment is the size of the
tile, minus the remainder of the width
    x_size = tile_xspace - (obj_w mod tile_xspace);
}

// The scale of each tile is the size divided by the
original size.
x_scale = x_size  tile_xspace;

/ Determines the y size and scales similarly to the
above.
if (obj_h < tile_yspace  2)
{
    y_size = 0.5  obj_h;
}
else
{
    y_size = tile_yspace - (obj_h mod tile_yspace);
}

y_scale = y_size  tile_yspace;

/ Iterates along the width and height of the object by
the size of the tile.
for (var i = 0; i < obj_w; i += x_size)
{
    for (var j = 0; j < obj_h; j += y_size)
    {
        var tile_x, tile_y;

        // Determines the x coordinate of the tiles
```

```
within the tile texture.
        if (i == 0)
            tile_x = 0;
        else if (i + x_size < obj_w)
            tile_x = tile_xspace;
        else
            tile_x = tile_xspace  2;

        // Determines the y coordinate of the tiles
within the tile texture.
        if (j == 0)
            tile_y = 0;
        else if (j + y_size < obj_h)
            tile_y = tile_yspace;
        else
            tile_y = tile_yspace * 2;

        // Adds a new tile to the scene based on the
parameters derived above.
        var new_tile = tile_add(bg_tiles_main, tile_x,
tile_y + tile_yoffset, tile_xspace, tile_yspace, i +
x, j + y, tile_layer);

        // Scales the newly created tile.
        tile_set_scale(new_tile, x_scale, y_scale);
    }
}
```

Using scr_define_tiles

Now that `scr_define_tiles` has been written, it can be placed within the **Create** events of the two platforms types.

The following code is how it should appear in `obj_basic_platform`:

```
/// Creates a set of tiles for basic platforms.
scr_define_tiles(sprite_width, sprite_height, 32, 32,
0, 1000005);
visible = false;
```

And this is how the script should be called in `obj_solid_platform`:

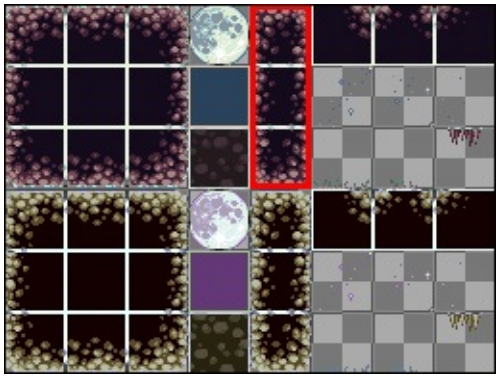
```
/// Creates a set of tiles for solid platforms.
scr_define_tiles(sprite_width, sprite_height, 32, 32,
96, 1000010);
visible = false;
```

The only difference between these two are the arguments used. Instances of `obj_basic_platform` will appear further back and different in color since they are offset vertically. Also, the `visible` property is set to `false` in both of these **Create** events. This means that upon creation, the different platforms, while maintaining their collision, will not be drawn. Because the tiles are being drawn though, it will appear as if the platforms have simply been replaced.

Drawing tiles in obj_moving_platform

The final issue that must be tackled with tiles is that instances of `obj_moving_platform` are still using the primitive art created in the previous chapter. Because this platform is moving and tiles are usually best left static, the tiles needed to represent this platform will be drawn using the **Draw** event instead.

Currently, instances of `obj_moving_platform` use the built-in function `draw_self` after drawing the path they are moving in. This will be replaced with the function `draw_background_general`. This built-in draw function requires many different arguments. In summary, it takes a section of a background resource and draws it to the specified position with scaling, rotation, color, and alpha blending applied. The next screenshot highlights the section of the tile texture that will be used in the **Draw** event:



The following code should replace `draw_self` in the **Draw** event of `obj_moving_platform`:

```
// Draws the moving platform using the existing
background instead of the original art.
draw_background_general(bg_tiles_main, 128, 0, 32, 96,
x, y + 32, 96  sprite_width, 32  sprite_height, 90,
cwhite, cwhite, white, cwhite, 1.0);
```

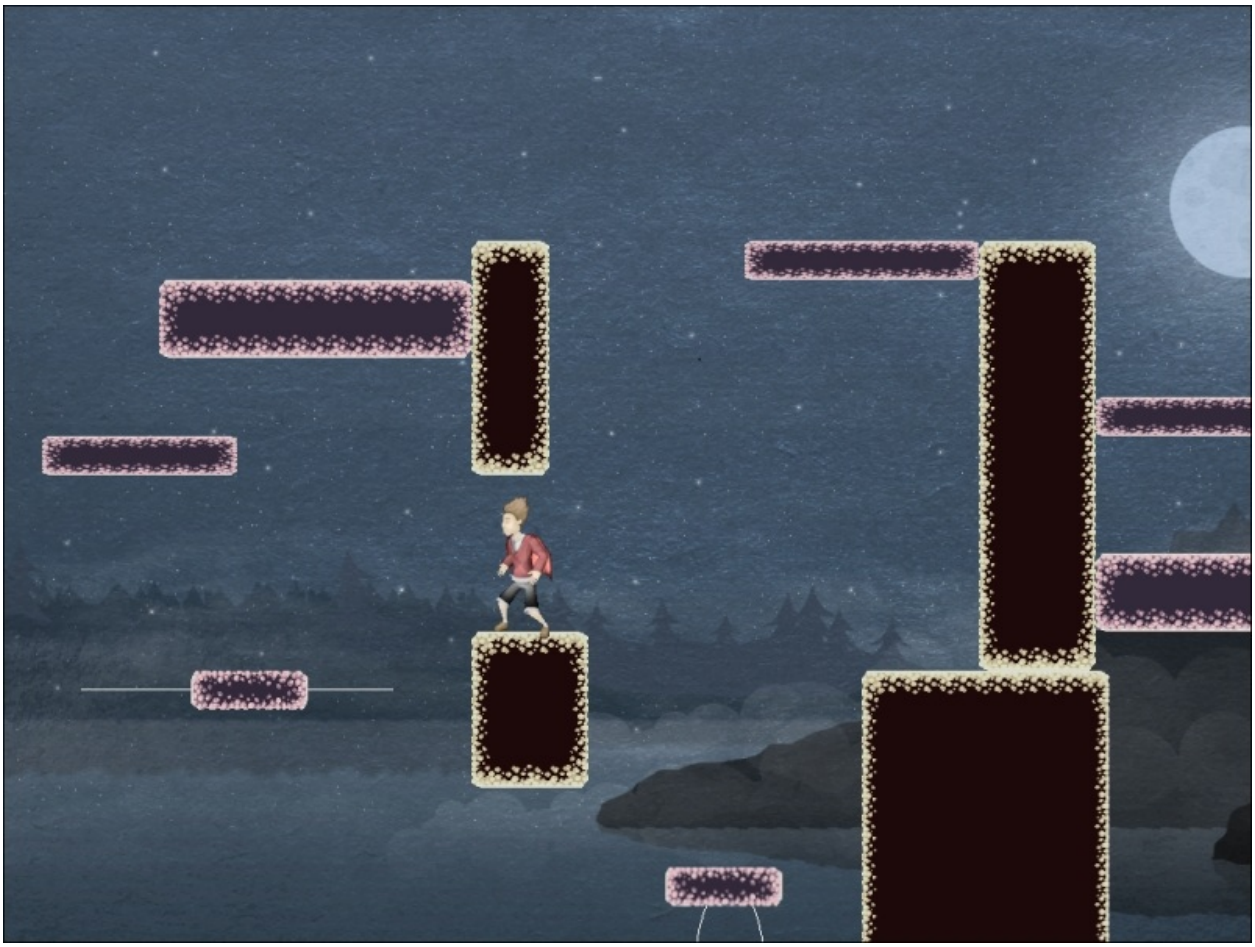
The previous code utilized 15 arguments, which are described as follows:

- `argument0`: This argument gives the background resource ID; in this case, `bg_tiles_main`
- `argument1`: This argument gives the left point of the rectangle within the background texture
- `argument2`: This argument gives the top point of the rectangle within the background texture
- `argument3`: This argument gives the width of the rectangle within the background texture
- `argument4`: This argument gives the height of the rectangle within the background texture
- `argument5`: This argument gives the `x` coordinate in which the background will be drawn
- `arguemnt6`: This argument gives the `y` coordinate in which the background will be drawn
- `argument7`: This argument gives the `x` scale
- `argument8`: This argument gives the `y` scale
- `argument9`: This argument gives the rotation in degrees
- `argument10-13`: This argument gives the blend colors at each corner

- `argument14`: This argument gives the alpha value

Though `32` is the width and `96` is the height of the rectangle being pulled from the background texture, this texture will be rotated, which is why the `x` scale is determined by dividing `96` by the sprite's width and the `y` scale by dividing `32` by the sprite's height. The drawn rectangle is also moved down 32 pixels (or its height) since the origin would technically lie at the lower-left of the rectangle instead of the upper-left, like the original sprite.

With the previous scripts implemented, Vlad should be followed by the camera, the background should move with the camera, and the programmer art for all platforms—both static and moving—should be replaced with the newly created tiles, giving the sense of a consistent environment, as shown in the following screenshot:



Summary

In conclusion, resources and scripts were written to track Vlad and create a better sense of environment. A camera was created using views. This camera followed the main character in the scene and allowed him to move past bounds of the room with the slight adjustment of some limit variables specific to the camera.

Then, backgrounds were discussed, and the solid blue background was replaced with a moonlit landscape that created atmosphere. Also, this background was moved with the camera, but at a different rate, making it feel as though it is far off in the distance and not just a backdrop stuck to the camera.

Finally, tiles were used to replace the simple, programmer art used to define the platforms in the previous chapter. Tiles were both placed through scripts during the static platforms' **Create** events as well as drawn during the moving platform's **Draw** event.

Through an improved environment, Vlad can now run and jump onto different platforms at this point in the game's development. Though the placement of these platforms can create an interesting challenge, there are no hazards, no enemies, no sense of danger in the game! In the next chapter, a variety of hazards and enemies that can damage Vlad—as well as some collectables that can heal him—will be created. The user interface will also be reviewed so that Vlad's health and a simple score can be tracked.

Chapter 9. Breaking Vlad – Pickups, Hazards, and Enemies

In previous chapters, views and backgrounds were discussed and an environment was built for Vlad to travel through while being tracked by a camera. This gave a little bit of depth to the game; however, the platformer is still rather simplistic, lacking the pressures and challenges commonly present in this genre's games.

There are dozens, if not hundreds, of ways pressure can be added. In this example, the concept of health will be the focus—components that decrease health, such as enemies, as well as those that can increase it. This topic will be covered in the following ways:

- A UI will be created to display score and health with the **Draw** and **Draw GUI** events
- Collectible items for increasing score and health will be created
- Vlad will be damaged through the creation of stationary hazards, enemies, and when falling offscreen

Tracking health with Draw and Draw GUI

Events have been discussed previously. In this chapter, these events will be used to display two parameters: the player's score and health. The score is being recorded and displayed as a way for the player to visualize their progress. Health is being tracked similarly, but unlike score, which increases, health will decrease. When it falls to `0`, the player will "lose" and have to start the level all over again.

Displaying UI with the Draw and Draw GUI events

To add this functionality, no new resources will need to be created; however, a **Draw** event and a **Draw GUI** event must be added to `obj_vlad`. An **Alarm** event will also be added to Vlad and some new variables will be added to his **Create** event. The score will be displayed using the **Draw GUI** event. It will be locked to the upper-left corner of the screen, somewhat out of the player's main focus, which should be Vlad. Meanwhile, the healthbar will be drawn above Vlad and will move with him. So why the alarm? Well, having the healthbar present at all times can end up being rather distracting, obnoxious even, so the alarm will be used to display this healthbar temporarily.

Creating a new font – fnt_score_text

Even though there will only be one font used in this chapter, it is still a good idea to create a new font resource so as to avoid conflict if additional fonts are needed. This font resource named `fnt_score_text` will use the defaults established when creating a new font resource, which includes using **Arial**, a font size of **12**, and a font range of **32 to 127**.

Setting up the Draw GUI event

The **Draw GUI** event is best used when graphical user interface elements are required to be drawn in consistent positions. In this case, the score will be kept in the upper-left corner of the screen. The following code should be added to the **Draw GUI** event of `obj_vlad`:

```
/// Draws a score at the upper left of the screen.

// Sets the color.
draw_set_color(c_white);

// Sets the horizontal alignment.
draw_set_halign(fa_left);

// Sets the vertical alignment.
draw_set_valign(fa_top);

// Sets the font
```

```
draw_set_font(fnt_score_text);

// Draws the text.
draw_text(0.04  view_wport[0], 0.04  view_hport[0],
"Score:  " + string(score));
```

In the previous code, basic `draw` parameters are set first: color, horizontal alignment, vertical alignment, and font. Then, `view_wport` and `view_hport` are multiplied by `0.04` so that the score text will be positioned four percent of the screen from the top and left. Using percentages allows for the score to be placed in relatively the same spot if the viewport's size is adjusted later.

Displaying health with the Draw event

In `obj_vlad`, the following code will be added to the **Draw** event:

```
/// Draws Vlad but only if he has health.

// Only draws Vlad if his health is greater than 0.
if (health > 0)
{
    // Draws Vlad.
    draw_self();
}

// Draws the health bar if the healthbar alarm is
active.
if (alarm[0] > 0)
{
    draw_healthbar(x - 50, y - 125, x + 50, y - 120,
health, c_black, c_red, c_green, 0, true, true);
}
```

First, the **Draw** event will only occur when the value of `health`, GameMaker: Studio's built-in global variable, is greater than `0`. The idea is that when the `health` variable is less than or equal to `0`, the player has lost and the character will no longer be visible since `draw_self` will not be executed.

Then, the value of `alarm[0]` is tested. If it is active, or greater than `0`, the

healthbar will be drawn in relation to Vlad's position using `draw_healthbar`. This healthbar will be centered above Vlad's head. It will use the black backdrop and an outline, gradating from green when the value of the player's health is `100` to red when the health value is close to `0`.

Updating the Create event

In the **Create** event of `obj_vlad`, health will be set to `100`, a new variable will be created, and `alarm[0]` will be set so that the healthbar can be seen on startup. The following block of code can be added to the end of the **Create** event script:

```
// Sets the Vlad's health to 100.
health = 100;

// If true, enemies can damage Vlad.
can_damage = true;

// Displays the healthbar on startup for about 3
seconds.
alarm[0] = room_speed * 3;
```

The `can_damage` variable will be discussed in the next section; this variable indicates whether or not the player can be damaged. Then, `alarm[0]` is set to the product of `3` and `room_speed`, which should make the healthbar be displayed for about 3 seconds when starting.

Arming the Alarm 0 event in obj_vlad

Now that the alarm is being set in the **Create** event, its script needs to be written, as shown in the following code:

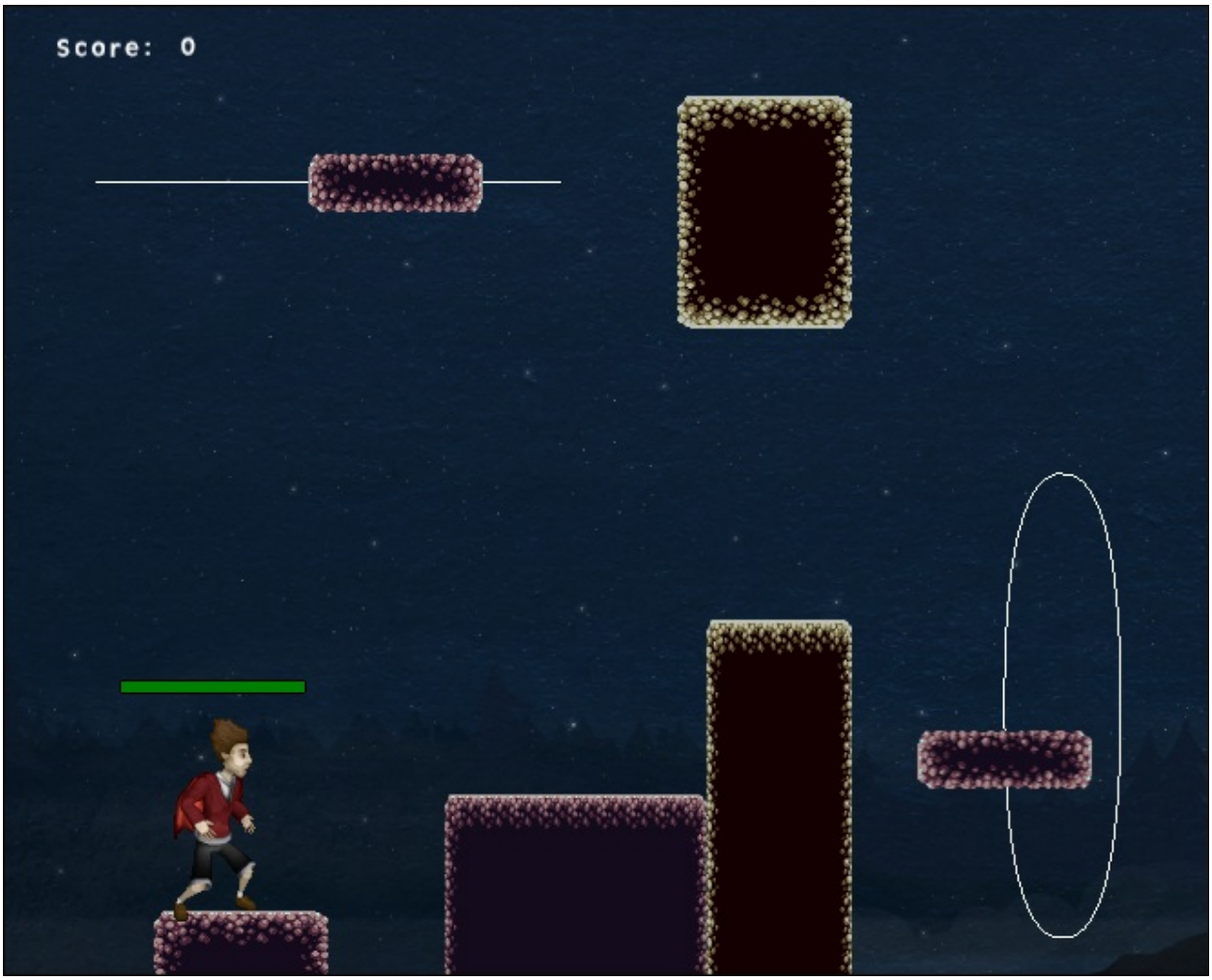
```
/// Alarm 0 script.

// Sets Vlad's alpha to 1.0.
image_alpha = 1.0;

// Allows the player to be damaged.
can_damage = true;
```

This simple script does two things. It first sets `image_alpha` to `1.0`. Then, `can_damage` is set to `true`. This is because, other than the **Create** event, the only time `alarm[0]` will be set is when the player is damaged. Two things will occur when Vlad is damaged: he will become semitransparent, signifying that he cannot be damaged, and `can_damage` will be set to `false`. These actions will be discussed in more detail in a future section in this chapter.

With these events created and their scripts written, the player should see the following scene when starting the game:



The white score text is on the upper-left corner of the screen and the health display is above Vlad's head. He is still opaque, though, so he can be damaged despite what occurs in the **Alarm** event.

In the next section, pickups will be introduced, one of which will increase Vlad's score while the other will increase his health.

Working with pickups

Power-ups, treasure, items, collectibles; **Pickups**, as they will be referred to in this text, go by a variety of names. In this game, there will be two types: one will simply increase the score while the other will give Vlad a boost in health. These sound like simple uses for pickups, but they can also act as helpful hints to the player, guiding them if they get stuck or helping them remember an area they may have already visited.

Gathering resources to create pickups

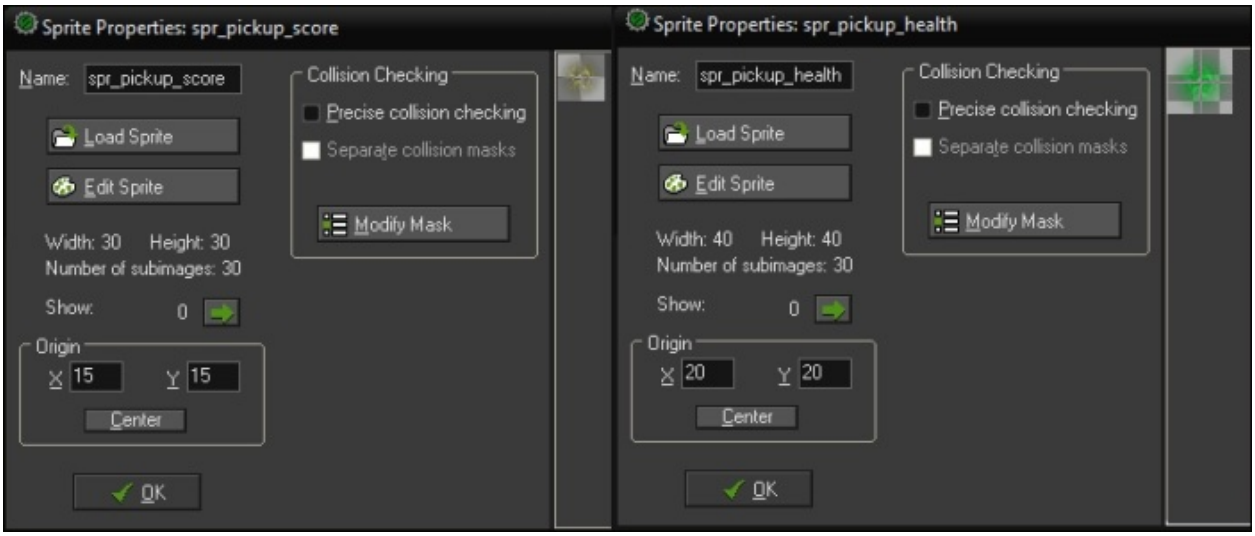
The pickups will require two sprite resources, two objects resources, and one script resource. The object resources will have only a **Create** event. The **Collision** events will be added to `obj_vlad` so it can interact with the pickups.

Initializing sprite resources for pickups

The sprite resources for the pickups should pop or stand out from the dark background previously established. They shouldn't distract too much but they should look enticing and inviting and definitely not dangerous. Because Vlad is a vampire, these two sprite resources will have a more magical, mysterious look to them.

The first sprite resource, which will be named `spr_pickup_score`, is of a golden, swirling orb, closely resembling a coin of sorts. The origin is centered at `15` horizontally and vertically. The mask is not updated.

The second sprite resource, `spr_pickup_health`, is similar to `spr_pickup_score` in that it is also an ethereal, spinning orb. It is slightly larger than `spr_pickup_score` in size, though, and green in color to match the color of a full healthbar. Due to the larger size, its origin is at `20` both horizontally and vertically. Its mask is also not updated. The details for `spr_pickup_score` and `spr_pickup_health` are detailed in the following screenshot:



Pickup object resources

After the sprite resources have been created, two object resources can be created. The first object, `obj_pickup_score`, will use `spr_pickup_score` as its sprite; the second object, `obj_pickup_health`, will use `spr_pickup_health`.

They will both have very similar **Create** events. In both **Create** events, visual code will be established. Then, some instanced variables will be created to help determine their effects. The following is the code for the events:

```
/// Sets up basic parameters

// Sets a random starting frame.
image_index =
irandom(sprite_get_number(sprite_index));

// Plays the orb spinning animation at half-speed.
image_speed = 0.5;

// The amount of points the player will earn when
collecting the orb.
points_earned = 100;

// The amount of health the player will earn when
collecting the orb.
health_earned = 0;
```

First, a random `image_index` object is set using `irandom` and by getting the number of subimages using `sprite_get_number`. This will establish visual variety between different pickups. The orb's spinning rotation will also be slowed down. Then, `points_earned` will be used to determine how many points are earned when Vlad collects instances of this pickup; `health_earned` will be added to the current value of the player's `health`. In `obj_pickup_score`, the `health_earned` value is set to `0`, but `health_earned` still needs to be defined for the script resource that will be defined soon.

The following is the **Create** event for `obj_pickup_health`:

```
/// Sets up basic parameters

// Sets a random starting frame.
image_index =
irandom(sprite_get_number(sprite_index));

// Plays the orb spinning animation at three-quarter-
speed.
image_speed = 0.75;

// The amount of points the player will earn when
collecting the orb.
points_earned = 200;

// The amount of health the player will earn when
collecting the orb.
health_earned = 10;
```

The idea is that the `obj_pickup_health` object should be placed less frequently in the level since the player not only earns back health, but gets a large increase in score as well. Also, this orb will spin a little faster since it has a higher `image_speed` value, making it more noticeable.

Script resource – scr_collect_pickup

So, before adding the **Collision** events to `obj_vlad`, a script resource to handle the event should be written. Because `obj_pickup_score` and `obj_pickup_health` are so similar, this script will handle either object being collected.

The first part of this script deals with increasing the score and health, as shown in the following code:

```
// argument0 -- the pickup collected.

// Awards the player points.
score += argument0.points_earned;

// Increases the player's health and shows the
healthbar.
if (argument0.health_earned > 0)
{
    // Increases the alarm to display the healthbar
since if false it is already being shown.
  alarm[0] = max(alarm[0], 2 * room_speed);


    // Increases health but makes sure it is less than
100.
    health += min(100, health +
argument0.health_earned);
}
```

First, `score` is increased by `points_earned` in `argument0`, which represents the recently collected pickup. Then, if the value of `health_earned` in `argument0` is greater than `0`, the value of `alarm[0]` is set to about 2 seconds using `room_speed * 2`. The `max` variable is used to make sure that if the `alarm[0]` array is already running, it won't be reduced by more than 2 seconds. The `health` value is then increased and `min` is used to make sure the value of `health` doesn't exceed `100`.

After this, the object is destroyed using the following `with` statement:

```
// Destroys the pickup.
with (argument0)
{
    instance_destroy();
}
```

The object is destroyed to prevent it from being picked up multiple times. A `with` statement is used because `argument0.instance_destroy()` is not a valid call. Shifting focus to the object represented by `argument0` using the

`with` statement will allow the object to be destroyed when the program is calling `instance_destroy`.

The following is the `scr_collect_pickup` script in its entirety:

```
// argument0 -- the pickup collected.

// Awards the player points.
score += argument0.points_earned;

// Increases the player's health and shows the
healthbar.
if (argument0.health_earned > 0)
{
    // Increases the alarm to display the healthbar
since if false it is already being shown.
  alarm[0] = max(alarm[0], 2 * room_speed);


    // Increases health but makes sure it is less than
100.
    health += min(100, health +
argument0.health_earned);
}

// Destroys the pickup.
with (argument0)
{
    instance_destroy();
}
```

Creating this script may seem unnecessary for these two pickups, but if more are desired, such as a large one that rewards 1,000 points or one that refills the player's health entirely, this script will be able to process that pickup as long as it defines `health_earned` and `score_earned`.

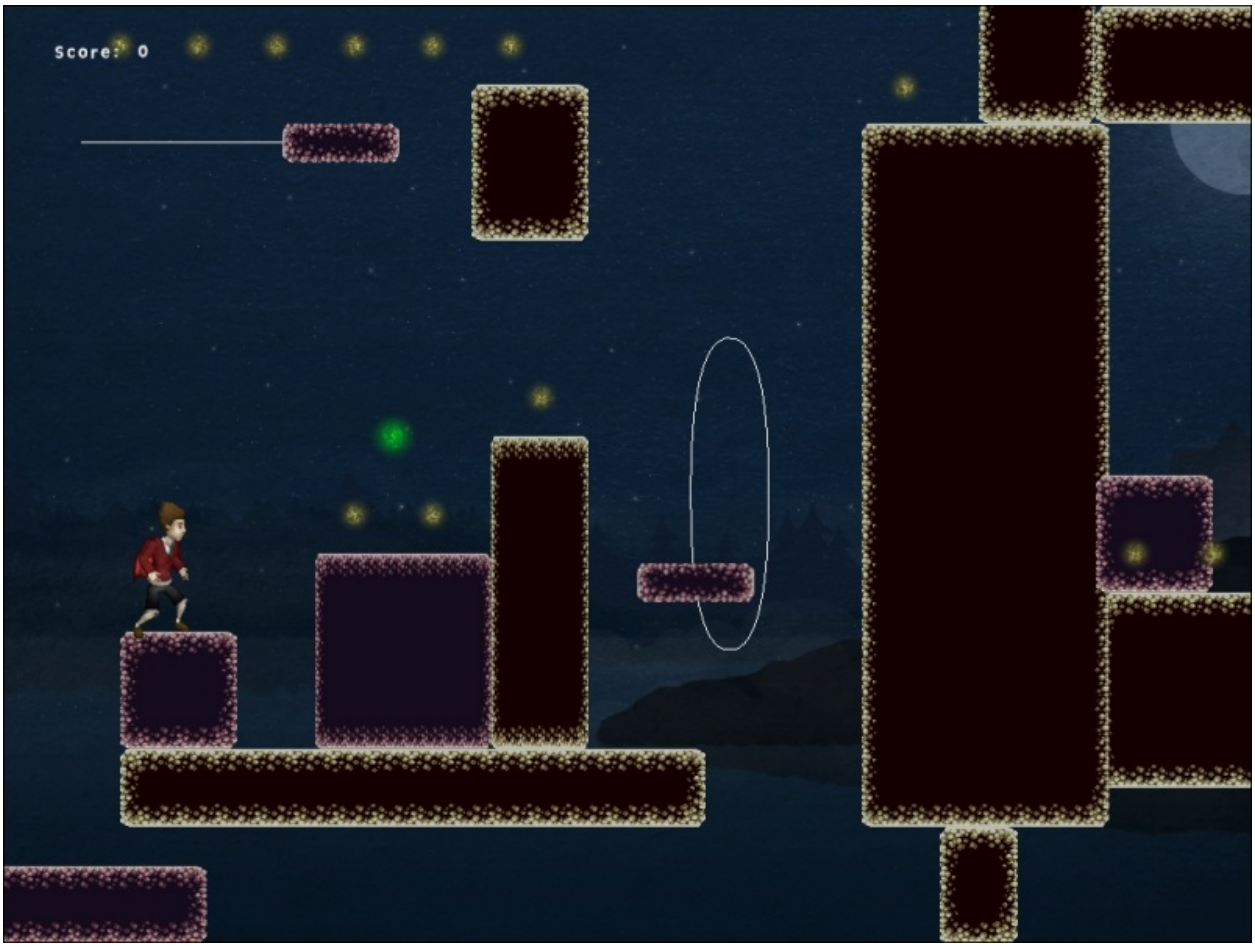
Colliding with obj_vlad

With the creation of the appropriate object resources and script resources, two **Collision** events can be added to `obj_vlad`. These **Collision** events should be linked to the respective objects: `obj_pickup_score` and `obj_pickup_health`. These are added to `obj_vlad`

because one object testing collision against multiple objects, instead of multiple objects checking collision for one, is more efficient. The script for these two objects will be identical, calling the previously written script resource, `scr_collect_pickup`, as shown in the following code:

```
/// Collects the pickup collided with.
scr_collect_pickup(other);
```

If written correctly, instances of `obj_pickup_score` and `obj_pickup_health` can now be placed in `rm_level1` and collected, as illustrated in the following screenshot:



Vlad can now collect orbs, but other than being breadcrumbs, they feel rather pointless. Up to this point, nothing has been able to damage or kill Vlad. This will be remedied in the next section with the introduction of a simple environmental hazard. Also, right now, Vlad can stand at the

bottom of the screen unlike many platforms where players—more often than not, it seems—fall into offscreen abysses.

Dying from hazards

Previously, pickups that help increase score and vitality were constructed, but nothing has been made that can actually cause Vlad harm. In this section, a simple hazard will be created. Then, a script will be written that will cause the player to "die" when they fall off the bottom of the screen.

Establishing a death state

Before creating any hazards, a new state should be created for Vlad: the death state. This will run when Vlad's health is set to a value less than or equal to `0` or if he falls below a specified value in the room.

A user-defined constant should first be created. In this instance, the constant will be named `state_death` and set to a value of `3`. This is so that **User Defined 3** will be launched during the **Step** event of `obj_vlad`.

Setting up two new events – User Defined 3 and Alarm 1

The next script will be triggered when the player's health runs out and enters the death state. This new state will be used to take control away from the player. The following script defines what should occur during the **User Defined 3** event:

```
/// DEATH STATE

// If the state was newly entered.
if (entered_new_state)
{
    // Prevents further damage.
    can_damage = false;

    // Sets the health to 0.
    health = 0;

    // Reduces the score by 5000, but without dropping
```

```
below 0.
    score = max(0, score - 5000);

    // Stops movement.
    hspeed = 0;
    vspeed = 0;
    gravity = 0;

    // Sets alarm 0.
    alarm[1] = room_speed * 3;

    // Prevents the state from being entered again.
    entered_new_state = false;
}
```

The previous script will only be executed if the death state was just entered. The `can_damage` value is set to `false` to prevent further damage and the `health` value is set to `0`; this is to prevent problems with `draw_healthbar` in the **Draw** event. Then, the value of `score` is reduced by `5000` points, but to prevent discouraging the player with a negative value, `max` is used to make sure the score doesn't fall below `0`. Then, all of the movement is stopped by setting `hspeed`, `vspeed`, and `gravity` to `0`. The value of `alarm[1]` is set to `3` times `room_speed`, about three seconds so that there is a slight delay instead of instantaneously restarting the scene, and finally `entered_new_state` is set to `false` to prevent this state from being entered repeatedly.

As for the **Alarm 1** event, the following lines of code are utilized:

```
/// Restarts the current room after the player has
died after a slight delay.

// Restarts the room.
room_restart();
```

This calls the built-in function `room_restart`, which will restart the current room. This is an appropriate punishment for losing in the room. Making it through the level without dying is the unwritten goal right now, so having to restart this mission after dying is acceptable. Also, using `room_restart` will reset the positioning of the character and enemies and quickly restore any collected pickups.

Gathering resources for hazards

A very common hazard is the spike. Spikes can be laid on the ground, against walls, or even hang like stalactites. When they're stationary, spikes can serve as interesting challenges by creating elements for players to dodge.

To create this new hazard, the following elements will be made:

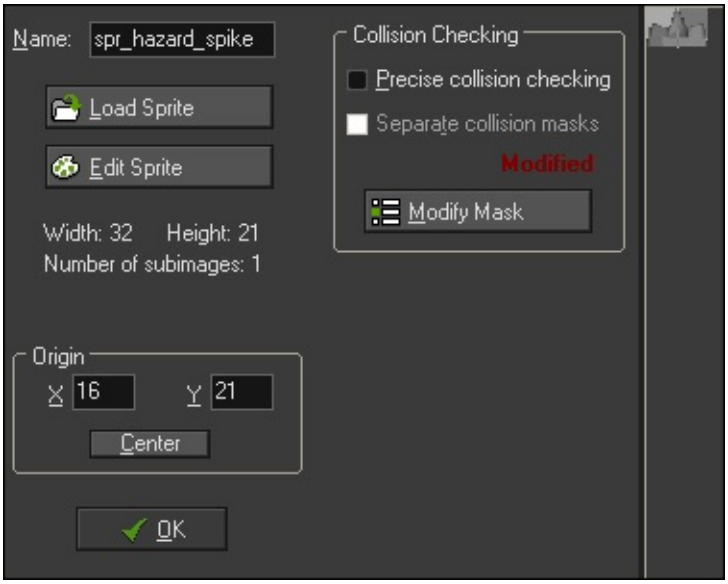
- A sprite resource: `spr_hazard_spike`
- An object resource: `obj_hazard_spike`
- A script: `scr_damage_vlad`
- A **Collision** event to `obj_vlad`

Creating the sprite and object resources

The artwork for the spike will be simple: a short, silver spiked triangle. It sounds more like something for werewolves than vampires, but the audience probably won't mind suspending more of their already suspended disbelief.

The resource, which will be named `spr_hazard_spike`, is 32 x 21 pixels. Its origin is centered horizontally at `16` and vertically at `21`. This will make rotating at its base easier.

As for the mask, it'll be set a little narrower than the entire sprite itself, `6` on the left and `25` on the right, allowing for a little tolerance for the player, which makes the most crucial part of the sprite register damage. The mask will fill the sprite vertically, however, with **Top** set as `0` and **Bottom** set as `20`. This sprite resource is illustrated in the following screenshot:



The object resource `obj_hazard_spike` has no events and uses `spr_hazard_spike` as its sprite. Interaction with this sprite will be added to `obj_vlad`, but first, a script will be written to account for this interaction.

Scripting scr_damage_vlad

The following script will be executed when `obj_vlad` collides with instances of `obj_hazard_spike`. Before creating this **Collision** event in `obj_vlad`, a new script resource, which will be named `scr_damage_vlad`, should be written.

This first portion of the following script will deal with setting Vlad's horizontal and vertical speeds when he is being damaged:

```
/// Damages Vlad.

/* arguments
  argument0 -- The object collided with.
  argument1 -- The amount of health to be subtracted.
  argument2 -- Should Vlad react based on the
difference between the player and the object or his
own speed.
 /

// Vlad can be damaged and if the object is visible...
if (can_damage && argument0.visible)
```

```
{
    // Vlad enters his jump state with a tiny hop
    entered_new_state = true;
    grounded = false;
    state_id = state_jump;

  if (vspeed >= 0)
    {
        vspeed = -5;
    }
    else
    {
        vspeed = 5;
    }

    // Changes the player's speed.
    if (argument2)
    {
        // Makes the player recoil away from the enemy
they collided with.
        hspeed = sign(x - argument0.x)  abs(hspeed);
    }
    else
    {
        // Simply reverses the player's horizontal
speed.
        hspeed = -1;
    }
```

Firstly, if `can_damage` is `true` and `argument0` (the object collided with) is visible, the actual collision actions will occur.

Then, Vlad is made to jump back, recoiling away from what damaged him. The `jump` state is entered, setting `state_id` to the user-defined constant `state_jump`. The `grounded` function is set to `false`, and `entered_new_state` is set to `true` so that the character will enter the jump state but with some changes. The `vspeed` value is checked and if it is greater than or equal to `0`, which means Vlad is either standing or descending, it is set to `-5`, making Vlad jump up. If he is ascending though, vspeed is simply set to `5`, making him immediately descend.

Finally, `argument2` determines how Vlad's `hspeed` variable is set. If it is `true`, the sign of the difference between Vlad and the other object's `x`

coordinate is determined. This is multiplied by the absolute value of Vlad's `hspeed` value to determine the direction in which Vlad should recoil, as if pushed by the hazard. If it is `false`, the value of Vlad's horizontal speed will be multiplied by `-1`, making him change direction by reversing it.

After altering Vlad's speed, variables concerning his health are altered as shown in the following code:

```
// Makes the player temporarily invulnerable to being
damaged and semitransparent.
    can_damage = false;
    image_alpha = 0.75;
    alarm[0] = room_speed * 3;

    // Reduces the player's health.
    health -= argument1;

    // If the player's health is less than 0, the
player will die.
    if (health <= 0)
    {
        state_id = state_death;
        entered_new_state = true;
    }
    }
```

First, `can_damage` is set to `false` to prevent Vlad from being damaged. Then, `image_alpha` is set to `0.75` so that Vlad appears to be semitransparent, which is a helpful way to show the player that Vlad is temporarily invulnerable. The value of `alarm[0]` is set to `room_speed * 3` so that the healthbar is shown for a few seconds as well. Then, the value of `health` is reduced by the value of `argument1`, and if the value of `health` is less than `0`, Vlad will enter the death state.

The following is the `scr_damage_vlad` script in its entirety:

```
// Damages Vlad.

/* arguments
  argument0 -- The object collided with.
```

```
  argument1 -- The amount of health to be subtracted.
  argument2 -- Should Vlad react based on the
difference between the player and the object or his
own speed.
 /

// Vlad can be damaged and if the object is visible...
if (can_damage && argument0.visible)
{
    // Vlad enters his jump state with a tiny hop
    entered_new_state = true;
    grounded = false;
    state_id = state_jump;

    if (vspeed >= 0)
    {
        vspeed = -5;
    }
    else
    {
        vspeed = 5;
    }

    // Changes the player's speed.
    if (argument2)
    {
        // Makes the player recoil away from the enemy
they collided with.
        hspeed = sign(x - argument0.x)  10;
    }
    else
    {
        // Simply reverses the player's horizontal
speed.
        hspeed = -1;
    }

    // Makes the player temporarily invulnerable to
being damaged and semitransparent.
    can_damage = false;
    image_alpha = 0.75;
    alarm[0] = room_speed * 3;

    // Reduces the player's health.
    health -= argument1;

    // If the player's health is less than 0, the
```

```
player will die.
    if (health <= 0)
    {
        state_id = state_death;
        entered_new_state = true;
    }
}
```

This script handles some cases that have yet to be discussed, but its original goal still remains: to create a reaction and reduce the player's health upon colliding with a hazard. The only step remaining is calling this function when `obj_vlad` collides with instances of `obj_hazard_spike`. This can be done by adding a **Collision** event to `obj_vlad` with the following code:

```
/// Damages vlad when colliding with spikes.
scr_damage_vlad(other, 10, false);
```

This will call the previously written script, reducing Vlad's `health` value by `10`, making him jump and reverse his horizontal speed upon collision.

Now that the spikes have been made, they can be placed around the level as shown in the following screenshot, causing damage to Vlad and adding an extra challenge for the player:



Falling off the screen into the abyss

So, now that the player can withstand damage from touching spikes, functionality will be added for the additional way in which the player can enter the death state: falling offscreen.

To achieve this effect, several changes will have to be made. The first will be to the **User Defined 2** event in `obj_vlad`, which is at the end of the script, when the sum of the player's `y` and `vspeed` values is greater than the value of `global.room_bottom`. This changed code is highlighted in the following code snippet:

```
if (y + vspeed > global.room_bottom)
{
    // Character is moved to the bottom of the room.
    y = global.room_bottom;
```

```
    // Speed and gravity are set to 0 to stop vertical
movement.
    vspeed = 0;
    gravity = 0;

    // Jump counter is reset.
    jump_count = 0;

    // The state is set to the death state.
    state_id = state_death;     // Specifies that the
state has been entered.
    entered_new_state = true;
}
```

Previously, `obj_vlad` would return Vlad to the idle state; however, now he will enter the recently written death state. The only problem now is that if this were to be tested, the player would die before Vlad actually leaves the view. A simple change to the **Creation Code** page of `rm_level1`, as shown in the following code snippet, will remedy this:

```
// Defines the bottom of the room
global.room_bottom = 950;
```

Previously, `global.room_bottom` was set to `800` so that Vlad could still be seen onscreen when reaching the bottom of the room and returning to the idle state. Since `spr_vlad_jump`, the sprite animation that should be playing when Vlad is in the jumping state, is `131` pixels tall, adding about `150` should be enough space so that when Vlad leaves the room, the death state will be entered and the necessary actions to restart the room are performed.

Now that spikes and the danger of falling offscreen have been implemented, a new type of hazard will be introduced: the enemy! In the coming sections, two simple enemies will be constructed to create even more danger and peril for the player.

Fighting the player with enemies

Enemies in platformer games such as the one being built in this book can come in a vast variety. There are some common themes that exist when discussing most enemies though, as shown in the following list:

- They can cause harm to the player
- They are usually not static, acting like moving hazards
- They can be killed by the player

Now, not all enemies will exhibit all of the behaviors previously listed, but a good majority of them do. In this text, the enemies created will focus on at least two of these items at a time, particularly the ability to harm as well as be killed by the player.

There will be two enemies created in total. A simple enemy that walks back and forth and another that will utilize GameMaker's built-in, simple pathing capabilities and fly towards the player.

Enemy 1 – mutant garlic

The first enemy that will be created is a mutant piece of garlic. Vampires and garlic don't get along most of the time, and if garlic could move, it'd probably be pretty dangerous—but not too intelligent. This being said, this anthropomorphic garlic enemy will simply walk back and forth on static platforms, changing direction when it either collides with a solid wall or the current platform it is walking on comes to an end. One of the most common ways to kill enemies in a platformer is to jump on its head, and this game won't deviate from that. Maybe Vlad is wearing anti-garlic footwear that allows him to do so.

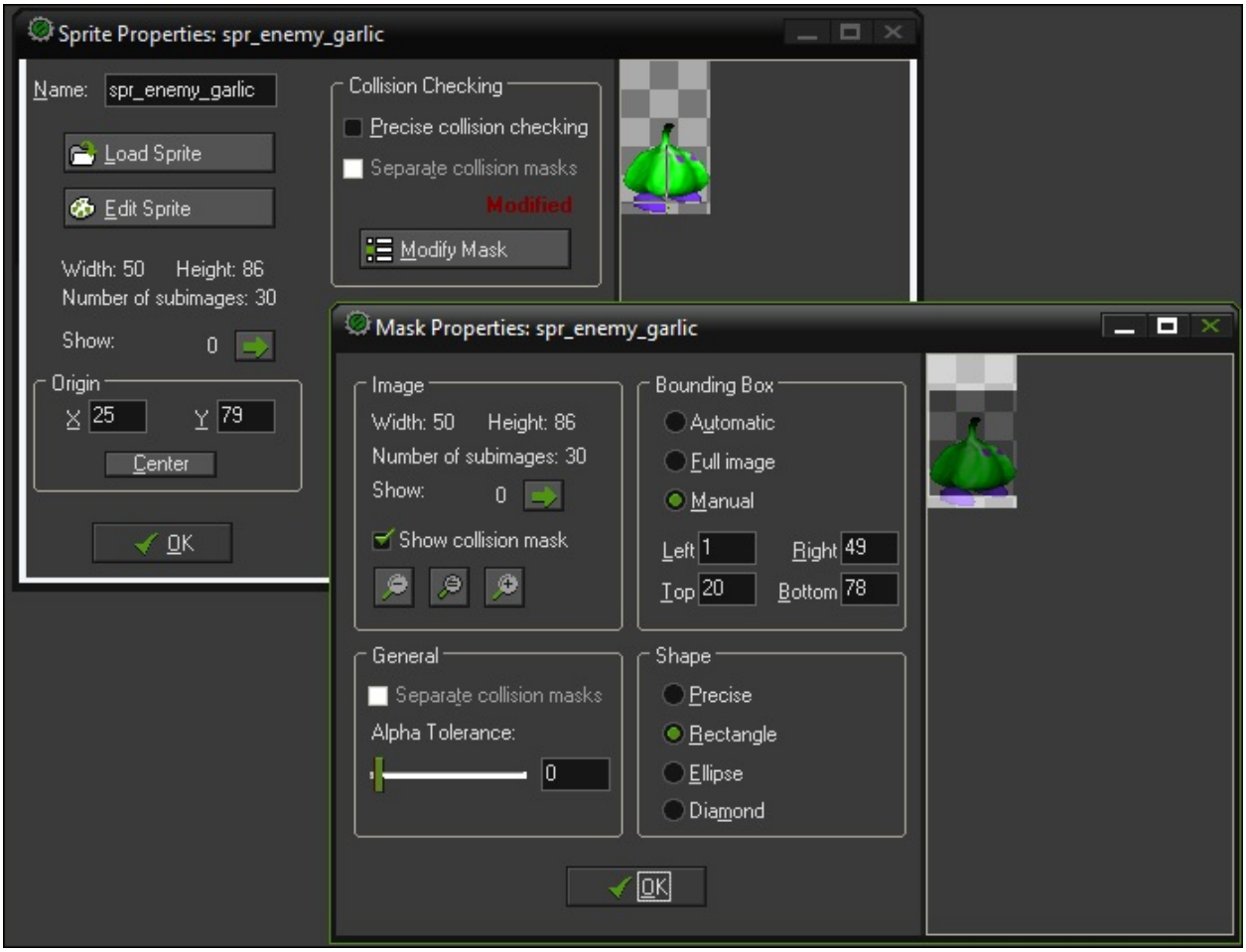
Gathering resources for garlic

To create this enemy, there will be several resources implemented as well as a script that will be used to create the other two enemies, as shown in the following list:

- `spr_enemy_garlic`: The sprite resource
- `obj_enemy_garlic`: The object resource
- `scr_collide_enemy`: The script executed when Vlad collides with any enemy

Sprite resource – spr_enemy_garlic

For the garlic, a 30-frame animation was created. The sprite is `50` x `86` pixels with its origin at `25` x `79`. The character playfully squashes and stretches as it moves, so a custom collision mask is used with `1` on the left, `49` on the right, `20` from the top, and `78` from the bottom. The following screenshot shows the parameters for `spr_enemy_garlic`:



Object resource – obj_enemy_garlic

After creating `spr_enemy_garlic`, a new object `obj_enemy_garlic` can be

created. This object has two events: a **Create** event and a **Step** event. The **Create** event will set up `obj_enemy_garlic` properly, and the **Step** event will be in charge of changing the direction of the enemy.

Scripting the Create event of obj_enemy_garlic

The following code is the **Create** event for `obj_enemy_garlic`:

```
/// Setup of the garlic character.

// Halves the animation speed.
image_speed = 0.5;

// Sets the animation speed.
hspeed = 2;

// How much is killing the enemy worth.
point_value = 100;

// If remains true, the object will be destroyed since
it isn't on a proper platform.
var destroy = true;

// Moves the character to the nearest platform down
vertically.
repeat (1000)
{
    if (!place_meeting(x, y + 1, obj_solid_platform)
&& !place_meeting(x, y + 1, obj_basic_platform))
    {
        y++;
    }
    else
    {
        destroy = false;
        break;
    }
}

// Destroys the instance.
if (destroy)
{
    instance_destroy();
}
```

The first several lines assign values to the instanced variables `image_speed`, `hspeed`, and `point_value`, which determine how many points the player will earn after the garlic is destroyed. Then, a local variable `destroy` is defined. In the second part, a `repeat` statement is used to try and make sure the garlic is standing on top of an instance of `obj_basic_platform` or `obj_solid_platform`. Neither `move_contact_solid` nor `move_contact_all` can be used because `obj_basic_platform` is not solid and instances of `obj_enemy_garlic` shouldn't be able to stand on pickups and other objects in the scene.

The `repeat` statement is essentially moving the character down by one unit with each repetition until either platform type is met. When a platform is met, `destroy` is set to `false` and the repeat loop is broken out of, but if the loop finishes and `destroy` is set to `true`, the instance is destroyed to prevent the creation of a floating garlic enemy.

Making garlic move with the Step event

Earlier, it was stated that this enemy would walk back and forth, changing direction when it determines that there is no more platform to walk on or when it collides with a wall. These actions will be taken care of in the following **Step** event code:

```
/// Makes sure the enemy changed direction when
reaching the end of a platform or wall

// Test to see if no floor is in the enemy's path
horizontally.
var no_floor = !place_meeting(x + sprite_width,y + 1,
obj_solid_platform) && !place_meeting(x +
sprite_width, y + 1, obj_basic_platform);

// Test to see if the enemy collides with a solid
wall.
var hit_wall = place_meeting(x + hspeed,y - 1,
obj_solid_platform);

if (no_floor || hit_wall)
{
    // Reverses the speed.
    hspeed = -1;
```

```
        // Flips the image horizontally.
        image_xscale = -1;
    }
```

The first two local variables, `no_floor` and `hit_wall`, use `place_meeting` to determine the enemy's relationship with instances of `obj_basic_platform` and `obj_solid_platform` in the room. If the character is offset by its `sprite_width` value—which will compensate for the value of `image_xscale`—and down one unit, but would not meet a platform of either type, `no_floor` will be `true`. On the other hand, if the sum of the enemy's `x` and `hspeed` values and its `y` position value minus `1`—since the enemy could be walking on a solid platform already, `hit_wall` will be `true`. If either of these are `true`, the `hspeed` and `image_xscale` values are both multiplied by `-1` so that the speed will reverse and the sprite will flip horizontally.

If the garlic enemy is implemented properly, it should be possible for the garlic enemy to be placed in the scene, and for now, it should be possible for the garlic enemy to harmlessly walk back and forth on different platforms, changing its direction when the described conditions are met. In the next section, a new script will be written that will determine how the game should react when Vlad collides with it.

Reacting upon collision with scr_collide_enemy

The following script is executed when Vlad collides with an enemy. This script has to determine whether the enemy was jumped on, thus killing the enemy, or Vlad is damaged:

```
/// Collision with an enemy.

// argument0 -- the enemy instance.

// If the character is moving down and collides from
the top.
if (vspeed > 0 && bbox_bottom < argument0.bbox_bottom)
{
    // Makes the character jump.
    entered_new_state = true;
    grounded = true;
```

```
            state_id = state_jump;

            // Destroys the instance of the enemy.
            with (argument0)
            {
                // Increases the score.
                score += point_value;

                instance_destroy();
            }
        }
        else
        {
            scr_damage_vlad(argument0, 10, true);
        }
```

In the previous block of code, two conditions are checked in the `if` statement. The vertical speed is compared to `0` to make sure the character is descending; the bounding box is also checked using `bbox_bottom` to make sure the character is above the enemy, as there are some conditions in which the player could miss the enemy.

If the enemy was collided with, Vlad will jump as if bouncing off it by setting `entered_new_state` and `grounded` to `true` and `state_id` to `state_jump`. Also, using a `with` statement, the score is increased by the value of `point_value`, which is set in the **Create** event, and the enemy is then destroyed using `instance_destroy`.

If the condition has failed, however, `scr_damage_vlad` is called. In this case, all enemies result in damage of `10` points. The `argument2` argument is `true` so that the positioning between the enemy and Vlad is used to determine the horizontal recoil.

Note

The `bbox_bottom` function, along with `bbox_right`, `bbox_left`, and `bbox_top`, represents a bounding box point of the object instance based on the current position of the sprite and its sprite collision mask.

Colliding with the player – the Collision event of obj_vlad

Now that `scr_collide_enemy` has been scripted, a **Collision** event can be added to `obj_vlad` that is similar to the one in `obj_pickup_health`. The script in this event should be as follows:

```
/// Calls the enemy collision script.
scr_collide_enemy(other);
```

This will be identical to the **Collision** events for the other enemies that will be created in this chapter as well.

Enemy 2 – the flying book

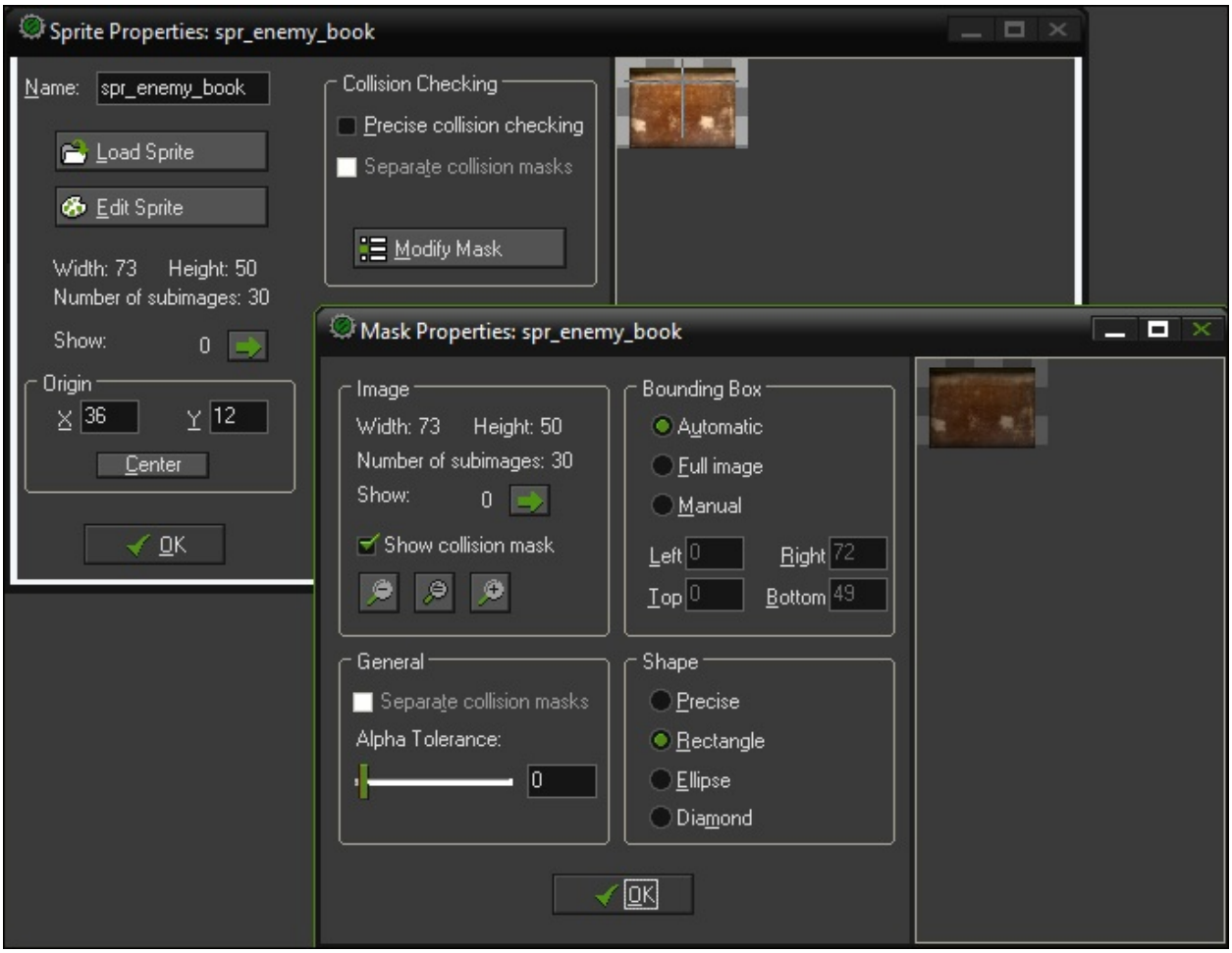
The garlic enemy is a rather simple enemy that requires some patience and timing to dodge or kill. The next enemy, an anthropomorphic, flying book won't be so easy. This enemy will check the distance between itself and Vlad, and when it's close enough, begin flying towards him! Because the script for colliding with enemies has already been written, implementing it will not be very difficult.

Gathering resources to create the flying book

Like the previously created enemy, a sprite resource and an object resource will need to be created; a corresponding **Collision** event will also need to be added to `obj_vlad`.

The sprite resource – spr_enemy_book

The following screenshot shows the parameters for the flying book, which uses the default mask. Its origin is centered horizontally at `36`, but is `12` units down from the top of the bounding box, near the book's binding. These parameters are shown in the following screenshot:



The object resource – obj_enemy_book

Like `obj_enemy_garlic`, this object resource will have a **Create** event and a **Step** event.

Initializing the Create event of obj_enemy_book

The following script will go in the **Create** event of `obj_enemy_book`:

```
// Slows the animation speed of the book down.
image_speed = 0.75;

// The point value awarded to the player.
point_value = 200;

// Tries to define the book's target.
```

```
target = instance_find(obj_vlad, 0);

// How far must the player be from the book before the
player is recognized.
distance_threshold = 250;
```

Similar to what we did with `obj_enemy_garlic`, instanced variables are defined in this **Create** event. The animation isn't played at full speed, and because these enemies will be more difficult to defeat, their `value for point_value` is higher. The unique variable `target` is set to the first instance of `obj_vlad`, while `distance_threshold` is set to `250`; this variable will be explained more thoroughly in the next section.

Anthropomorphizing the book in the Step event

The following is the **Step** event's script for `obj_enemy_book`:

```
if (distance_to_object(target) < distance_threshold)
{
    mp_potential_step_object(target.x, target.y - 50,
3, obj_solid_platform);
}
```

In the previous code, `distance_to_object`, a built-in function, is used to determine the distance between the book and the target. If this distance is less than the value of `distance_threshold`, the built-in function `mp_potential_step_object` is executed.

The object `mp_potential_step_object` belongs to a series of built-in functions used to help objects move along a path to a specified goal. It uses the following four arguments:

- `argument0`: This is the `x` coordinate to approach the goal
- `argument1`: This is the `y` coordinate to approach the goal
- `argument2`: This is the speed at which the object approaches the target
- `argument3`: This is an object resource type to avoid

The value of `target.x` is used as is, but then `50` is subtracted from the

value of `target.y` so the book moves towards the center of Vlad instead of toward his feet. The third argument means the book will move at a speed of `3`; however, using this function will not set the speed variables. Finally, using `obj_solid_platform` will make it so the book tries to avoid moving into instances of solid platform, making it appear obedient to the rules the player's character has to obey.

Note

The `mp` functions are a great way to create simple artificial intelligence, as the book will exhibit, but are rather complex functions with a lot of unique nuances. More about these functions can be found in GameMaker: Studio's help contents.

Finally, like `obj_enemy_garlic`, a **Collision** event must be added to `obj_vlad`; this **Collision** event will have the same exact script as `obj_enemy_garlic`: `scr_collide_enemy(other);`. If the **Collision** event is implemented properly, two types of enemies can now be placed in the scene to make Vlad's journey to the goal more perilous, much like the level shown in the following screenshot:



Summary

In this chapter, resources were created to develop pickups, hazards, and enemies for the player. Players can now increase their score and health by colliding with these objects. The player can also be harmed by interacting with stationary spikes and falling off the screen. Simple enemies were also created using predictive movement and built-in artificial intelligence functionality.

Right now, the player can traverse this environment, creating it from the lower-left of the scene to the lower-right, jumping, dodging hazards, and attacking enemies. The main component really missing now is feedback as there is very little. There isn't even really a hint that the player has achieved a goal. In the next chapter, feedback for a variety of aspects will be added, reviewing some concepts from Chapter 4, *Juicy Feedback – Aural and Visual Effects*, and exploring new ones.

Chapter 10. GOAL – Timelines and Feedback Review

In the previous chapter, enemies and pickups were created for the player to interact with in the platformer game. Now, many features exhibited by other games of this genre have been implemented; however, there is still no final goal for the player in the current room.

Over the course of this final chapter, a goal will be created so that the player has something to ultimately strive for in each room. Feedback and juiciness—similar to that introduced in Chapter 4, *Juicy Feedback – Aural and Visual Effects*—will also be added to the game through the use of the following:

- Particle effects
- Sound effects
- Timelines

GOAL!

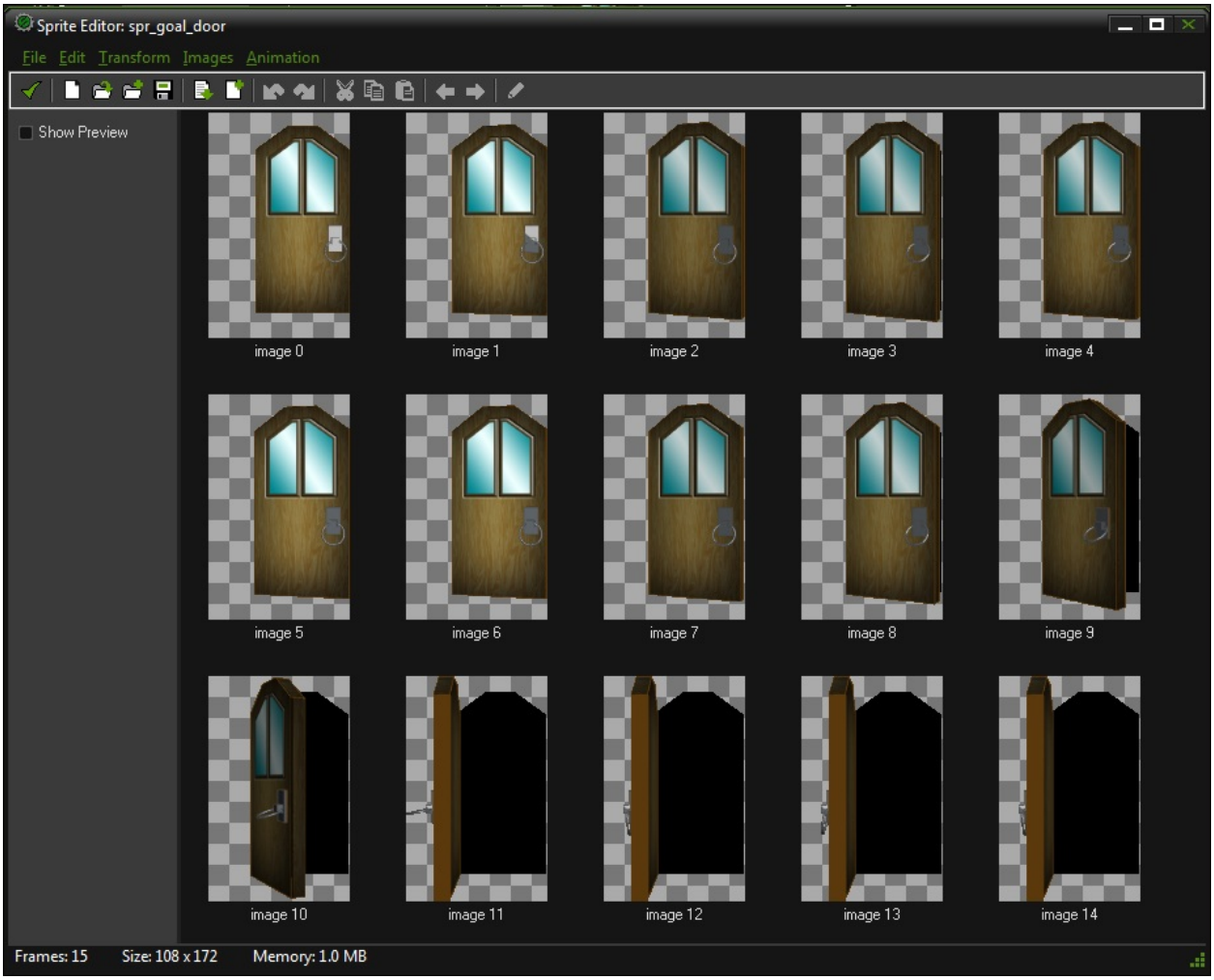
In platformers, there can be many types of goals: collecting a certain number of objects, defeating a specified number of enemies or a boss, or simply getting from point A to point B. In this game, having already established that there is some destination in the distance that the player may be trying to get to, the goal will be a simple marker that, when collided with, will take the player to the next room.

Gathering resources for creating the goal

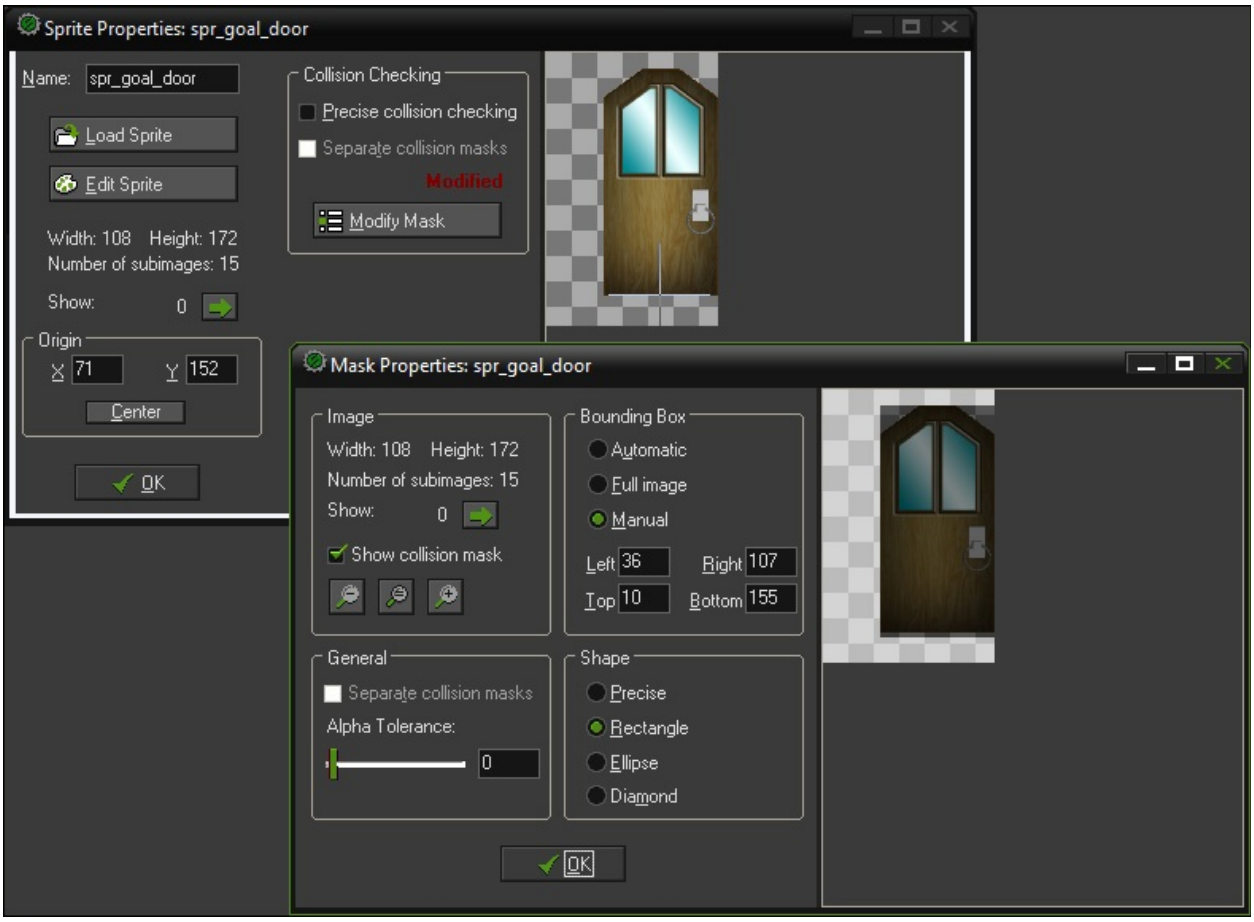
The goal in this game will be a mysterious, magical door that, when opened, will take Vlad to a new room. This goal will require a new sprite resource and a new object resource.

Sprite resource – spr_goal_door

The animation used for `spr_goal_door` is of a door opening, leading to a mysterious blackness, as demonstrated in the following sprite sheet:



The mask and origin are aligned according to the first subimage. Then, origin is centered to the bottom-center of the door at `71` and `152`. The mask is rectangular, covering the entire door on the first frame, its **Left** at `36`, **Right** at `107`, **Top** at `10`, and **Bottom** at `155`. The mask and origin settings for this sprite are demonstrated in the following screenshot:



Object resource – obj_goal_door

A new object resource named `obj_goal_door` will use the recently created sprite resource `spr_goal_door`. For now, it will only exhibit a **Create** event. Then, similar to what we did to create pickups and enemies, a **Collision** event will be added to `obj_vlad` when Vlad collides with instances of this object. The **Create** event script, as shown in the following code snippet, stops the animation from playing at the start of the game:

```
/// Defines variables for the goal door.

// Stops the animation.
image_speed = 0;
```

Once Vlad collides with the door, several actions will occur. First, Vlad will be awarded some points, which will act as a simple reward given to the player for completing the challenges in the room. A simple equation

will be used so that players that complete the challenges in the room with more health are awarded more points. Then, Vlad will go to the next room or, if he is in the last room, he will go back to the first room. The code for this is as follows:

```
/// Action that occurs when the door is collided with.

// Score is increased by remaining health.
score += 50 * health;
// Goes to the next room unless the player is in the
first room.
if (room == room_last)
{
    room_goto(0);
}
else
{
    room_goto_next();
}
```

The `room` variable is a built-in variable that represents the ID of the room that the player is currently in; meanwhile, `room_last` is the ID of the last room in the game. If these IDs match, the player will be sent to the first room using `room_goto` with `0` as its main argument; otherwise, `room_goto_next` will be used. This function will take players to the next room assuming there is a next room to go to.

Once the previously stated door is implemented, it can be placed in `rm_level1` or any room that has been built. In the following screenshot, the door is placed on a platform to the lower-right corner of `rm_level1`:



Now, when the door is collided with, the player should travel to the next room, or if a new room has not been created, restart the current room. These results, however, are rather jarring as they immediately catapult the player to their destination. This harsh transition is confusing because the player isn't given any time to breathe and there is a lack of important feedback. In the next section, **timelines** will be introduced, which will allow for sequences of scripts to be triggered.

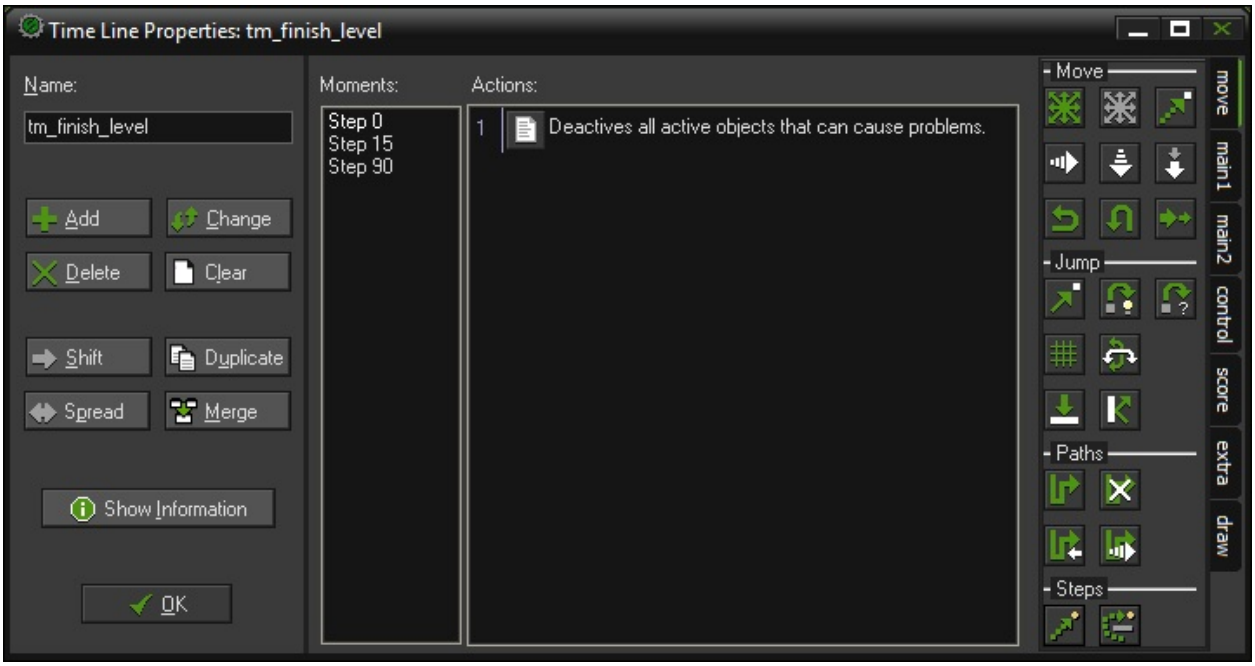
Introducing timelines

Previously, alarms were used to trigger a script after a short delay. Now, what if you want to execute several different scripts in a sequence? Something like the following block of code could be used within an **Alarm** event:

```
switch (alarm_state)
{
  case 0:
    score += 10;
    alarm_state = 1;
    alarm[0] = 10;
    break;
  case 1:
    player_text = "You win";
    alarm_state = 2;
    alarm[0] = 120;
    break;
  case 2:
    room_goto_next();
    break;
}
```

The previous code, however, could become difficult and confusing to edit if many different states are required.

This is where timelines come in. Timeline resources allow the user to define a series of scripted events or **moments** to trigger in a sequence. The following screenshot shows the **Timeline Properties** window:



There are eight buttons in the previous window that perform different actions, as described in the following list:

- **Add**: This button adds a moment to the timeline. Once the moment is added, scripts can be added to this moment just as with an event in an object resource.
- **Change**: This button changes the time step or the moment at which the selected timeline event will occur.
- **Delete**: This button deletes the selected timeline moments.
- **Clear**: This button clears the entire list of timeline moments.
- **Shift**: This button shifts all of the moments within a specified range for a certain number of steps.
- **Duplicate**: This button is similar to **Shift**, but it creates new moments in the timeline.
- **Spread**: This button spreads out a range of moments across the timeline by adding or removing time using a specified percentage.
- **Merge**: This button combines the moments within a specified range.

Using timelines

Timelines are similar to alarms in that applying a timeline to an object does not involve a function, but instead involves setting several variables.

These variables are explained in the following code sample:

```
// The timeline index associated with the object.
Defaults to -1.
timeline_index = tm_finish_level;

// The position of the timeline in steps.  Defaults to
0.
timeline_position = 5;

// Determines if the timeline will loop (true) or not
(false).
timeline_loop = true;

// The speed at which the timeline runs.  Defaults to
1.
timeline_speed = 0.5;

// Determines if the timeline is running (true) or not
(false).
timeline_running = true;
```

Overall, setting up a timeline and applying it is simple and quick, requiring the adjustment of only five built-in, instanced variables.

Note

There are a few built-in functions associated with timelines; however, because dynamic code creation has been deprecated, dynamic timeline creation is impossible. The only useful timeline function is `timeline_exists`, which can determine whether or not a specific ID represents an actual timeline resource.

Gathering resources for integrating the timeline

Now that timelines have been introduced, they will be used to create a delay when the player reaches the goal instead of instantly taking the player to the next room. Before creating any timeline resources, though, other resources as well as updates to the scripts of `obj_goal_door` will be created. The following list includes all of the updates that will be added to the game:

- A new font resource: `fnt_large_text`
- The addition of the **Draw**, **Draw GUI**, and **Animation End** events to `obj_goal_door`
- A new script resource: `scr_deactive_instances`
- A new timeline resource: `tm_finish_level`

Font resource – fnt_large_text

The messages that will be displayed to the player after completing the level need to be noticeable, so a large font will be created. To match the creepy theme of the game being established, a similar creepy font can be used. Readability is important, but because this isn't important information, a more fun, thematic font can be used. The font resource `fnt_large_text` uses the font type **Chiller** with a size of `64` and a font range of `32 to 127`. These settings are displayed in the following screenshot:



Creating and updating events for obj_goal_door

Before applying the timeline to `obj_goal_door`, several scripts should be updated, so the messaging can occur properly. The first step is the addition of some new variables in the **Create** event of `obj_goal_door`:

```
// Boolean that determines if the large text should be
drawn on screen.
show_screen_text = false;

// The text that will be displayed on screen by the
door.
screen_text = "";

// The number of items to draw while the game is
"paused".
draw_count = 0;
```

The first variable, `show_screen_text`, is a Boolean that will determine whether or not to execute the functions associated with drawing the text onscreen. Then, `screen_text` is the string that actually represents this text. Finally, `draw_count` is a variable that will be brought up when discussing the **Draw** event of `obj_goal_door` and the soon-to-be-constructed script `scr_deactive_instances`.

Using the Animation End event

Right now, the door is animated, but the scene moves so quickly that it is never seen. One goal of using the timeline is to play this animation, but of course it needs to stop playing at the end; this can be achieved by using an **Animation End** event, which is also used within `obj_vlad` to loop animations. The script for this is provided as follows:

```
/// Stops the animation on the last sub-image.
image_speed = 0;
image_index = image_number - 1;
```

By setting `image_speed` to `0`, the animation will no longer play, and setting `image_index` to the value of `image_number - 1`, the `obj_goal_door` object will stop on the final frame.

Drawing congratulatory text with the Draw GUI event

Since `obj_goal_door` can be placed in various locations, a **Draw GUI** event can be used to make sure the placement of text is consistent throughout. The following script will draw the text, making sure to set all of the appropriate parameters:

```
/// Draws the message text.

// If the messaging text should be drawn...
if (show_screen_text)
{
    // Sets the font.
    draw_set_font(fnt_large_text);

    // Sets the horizontal alignment.
    draw_set_halign(fa_center);

    // Sets the vertical alignment.
    draw_set_valign(fa_middle);

    // Draws a black version, offset for a drop-shadow
effect.
```

```
    draw_set_color(c_black);
    draw_text(view_wview[0]  0.495, view_hview[0]
0.495, screen_text);

    // Draws a white version centered to the view.
    draw_set_color(c_white);
    draw_text(view_wview[0]  0.5, view_hview[0]  0.5,
screen_text);
}
```

In the previous block of code, `show_screen_text` is first tested. This Boolean is used to prevent unnecessary `draw` calls. In this case, the amount of performance improvement is nominal, but avoiding unnecessary `draw` calls is a good, efficient habit to start practicing. Then, the font and alignments are set using built-in functions. Finally, the text is drawn twice: first a black version and then a white version. These are placed by multiplying the width and height of the view by different values, in this case, `0.495` and `0.5`. This can create a simple drop-shadow effect and make text more readable.

Deactivating objects with scr_deactive_instances and the Draw event

Before creating the timeline, another issue has to be addressed. This issue involves delays in games. Sometimes, when instituting delays, bugs and glitches can occur. For example, if Vlad were to reach the door, but a 3-second delay were to be inserted, and then he is killed by an enemy within that short period of time, what would happen? Well, for one, the player would probably be frustrated, thinking they had completed the level only to die.

There are a lot of ways to prevent this. Some games will disable all enemies onscreen; some will disable player's control on the game. In this case, both will be done! A moment of pause will be simulated, freezing all elements that could potentially damage the player while the instance of `obj_goal_door` opens.

Freezing instances with scr_deactivate_instances

To freeze all the instances of a type, two things have to be done. First, the instances have to be deactivated, which can be done with the built-in function `instance_deactivate_object`. Then, since deactivated objects are not rendered by default, information about the deactivated objects should be stored so that they can be drawn. The following new script `scr_deactivate_instances` will do just that:

```
// argument0:  the object index to look for.

// Finds the number of instances in the scene.
var ins_num = instance_number(argument0);
repeat(ins_num)
{
    // The instance at 0 is always since the number
changes as instances are deactivated.
    var instance = instance_find(argument0, 0);

    //The sprite indices, image_index, and positions
are saved.
    draw_items[draw_count, 0] = instance.sprite_index;
    draw_items[draw_count, 1] = instance.image_index;
    draw_items[draw_count, 2] = instance.x;
    draw_items[draw_count, 3] = instance.y;

    // The number of objects drawing is increased.
    draw_count++;

    // The object is deactivated.
    instance_deactivate_object(instance);
}
```

First, `instance_number` is executed using `argument0`, which represents the object resource ID of the type of object that will be deactivated. This returns a value that will use the `repeat` statement that follows in the code. Then, each type of the provided instance is found using `instance_find`. The second argument is always `0`; as objects are deactivated, the number of instances in the scene decreases. Using `0` will ensure that an instance is returned until they are all deactivated.

This function will be called exclusively by `obj_goal_door`, so first, `draw_items` is set in a two-dimensional array, the first index being `draw_count`. The other four values represent the sprite information and positioning of the instance that will be deactivated and are defined. This is followed by incrementing `draw_count` and finally deactivating the instance.

> **Note**
>
> Deactivation is not the same as deletion. Objects that are deactivated can be reactivated using `instance_activate_object` or `instance_activate_all`. Also, parameters of deactivated instances can still be accessed as long as their ID is stored. If the room ends, however, deactivated instances, even if they're set to be persistent, will be deleted.

Drawing deactivated instances

To draw the instances deactivated using `scr_deactivate_instances`, a rather straightforward **Draw** event will be added to `obj_goal_door`. This script will first draw the door and then all deactivated instances, as shown in the following code:

```
/// Draws the goal door and all deactivated instances.

// Draws the door.
draw_self();

// Draws the deactivated instances.
for (var i = 0; i < draw_count; i++)
{
    draw_sprite(draw_items[i,0], draw_items[i, 1],
    draw_items[i, 2], draw_items[i, 3]);
}
```

The door is first drawn using the familiar built-in function, `draw_self`. Then, using a `for` statement, each deactivated item is drawn using

`draw_sprite`. Now that the appropriate scripts and events have been established, the timeline for this goal sequence can be created.

Creating and applying the timeline

Having created the necessary events and scripts, the timeline resource that will be used by `obj_goal_door` can be created. This timeline resource will be named `tm_finish_level`. Timelines can be created by navigating to **Resources** | **Create Timeline** or by pressing *Shift* + *Ctrl* + *T*. This timeline will have three moments.

Step 0

When the timeline starts, it will immediately deactivate all instances that can cause problems in the room with the following script:

```
/// Deactivates all active objects that can cause
problems.
scr_deactivate_instances(obj_enemy_garlic);
scr_deactivate_instances(obj_enemy_book);
scr_deactivate_instances(obj_vlad);
```

All enemies and the player are deactivated using `scr_deactivate_instances`. This will make all elements that can damage the player, including the character himself, appear frozen while other objects, such as the animated pickups, will continue to move.

Step 15

The next moment is at step 15, which will last for about a half of a second. This moment will set the necessary variables to display text, as shown in the following code:

```
/// Sets and displays the screen text to be shown.
show_screen_text = true;

// Uses a different message if the current room is the
final room.
```

```
if (room == room_last)
{
    screen_text = "Congrats!  You made it home!#Let's
do it again!";
}
else
{
    screen_text = "Almost there!#Onto the next room!";
}
```

In the previous code, `show_screen_text` is set to `true` and then `screen_text` is set depending on the room index, using a unique value if the player is in the last room.

Step 90

The third and final moment will happen about 3 seconds after the timeline has started., and as this is the final moment, the following script will actually take the player to the next room:

```
/// Goes to the next room or to the first room if all
the levels have been completed.
if (room == room_last)
{
    room_goto(0);
}
else
{
    room_goto_next();
}
```

Look familiar? It should, as it contains a portion that was originally scripted within the **Collision** event of `obj_vlad` for `obj_goal_door`. That event will now be updated so that when it's collided with, the timeline will actually be assigned and run.

Applying tm_finish_level

Within the **Collision** event of `obj_vlad`, the following script will replace the `if` statement now used in step 90 of `tm_finish_level`:

```
// Applies the timeline to the object collided with,
in this case, the door.
with (other)
{
    // Checks to make sure a timeline is not already
assigned.
    if (timeline_index < 0)
    {
        // Assigns the timeline.
        timeline_index = tm_finish_level;

        // Makes sure it starts at the beginning.
        timeline_position = 0;

        // Makes sure the timeline doesn't loop.
        timeline_loop = false;

        // Runs the timeline.
        timeline_running = true;

        // Plays the door animation.
        image_speed = 0.5;
    }
}
```

Using a `with` statement, timeline-specific parameters are set for `obj_goal_door`. The `timeline_index`, `timeline_position`, and `timeline_loop` objects are all set. Setting `timeline_running` to `true` is what actually starts the timeline. Finally, `image_speed` is set to `0.5` to actually play the door-opening animation.

If properly set up, the door should swing open when the player collides with it, freezing enemy instances and Vlad. Then, a message should be displayed as demonstrated in the following screenshot:


Almost there!
Onto the next room!

Reviewing polish, feedback, and juiciness

As discussed in previous chapters, feedback is extremely important whether it's through messages, sounds, or effects. For the remainder of this chapter, techniques for feedback will be reviewed as they are a vital part of polishing any game. In this final portion, pickups and enemy interaction will be the focus.

Gathering resources to play sounds

Several sound resources will be created for use in this final portion, and they are as follows:

- `bgm_level_one`: This is a background music track. It can either be creepy to help continue establishing the creepy theme, or it can be something that serves as a strong juxtaposition to the current theme
- `snd_damaged`: Executing this produces a sound of pain—a grunt or moan that will be played when enemies or the player are damaged
- `snd_pickup_collect`: Executing this plays a quick, whimsical, magical sound when pickups are collected

Built-in functions will be used to play music as well as show how the same sound can be used and played at varied pitches to get a different effect without having to create a wide variety of sounds.

Playing music with scr_play_music

Music is an important element in games, adding atmosphere and sometimes breaking yawn-producing silence. It's important to know whether or not the music being played in a game should be playing continuously when changing rooms, levels, or states. In this case, the music in a level should play continuously unless Vlad travels to a new room that calls different music. The following script, created as a script

resource named `scr_play_music`, will be called by levels on their **Creation Code** page to make sure the right song is being played:

```
/// Plays a specified track but only if it isn't
playing already.

// argument0:  Sound Resource id of the music to be
played.
// argument1:  Should the music loop?

// Plays music if nothing is playing.
if (!audio_music_is_playing())
{
    // Sets the global music id.
    global.music_id = argument0;

    // Plays the music.
    audio_play_music(global.music_id, argument1);
}
else
{
    // If music is playing but not the specified
music.
    if (global.music_id != argument0)
    {
        // Sets the global music id.
        global.music_id = argument0;

        // Plays the music.
        audio_play_music(global.music_id, argument1);
    }
}
```

In the first portion of the code, the built-in function `audio_music_is_playing`, which returns a Boolean, is executed. If `false` is returned, a global value named `music_id` is created and set to `argument0`, which is the sound resource ID of the music that is going to be played in the game. Then, using `audio_play_music`, music is played using `argument1` as a Boolean specifying whether or not the music should loop.

If music is being played, but the value of `global.music_id` is not equal to the requested sound resource ID, this new track is played. The reason this is put into a separate `if` statement is that if `global.music_id` is

checked before being set, a runtime error exception will occur. Again, it is very important that variables—global, local, or otherwise—be set before trying to access them.

The script can then be placed at the end of the **Creation Code** page of `rm_level1` with the following:

```
scr_play_music(bgm_level_one, true);
```

This will play the music at the beginning of the level, but when it is restarted due to the player failing to complete the level, it will continue to play.

Reviewing the obj_particle_manager object

Before using the other two sound resources created, the particle manager created in Chapter 4, *Juicy Feedback – Aural and Visual Effects*, will be used once again. The particles created in that chapter will not be re-used, but the concept of having one object manage the particle will remain.

> ## Note
>
> The `obj_particle_manager` object was imported from the project created in Chapter 4, *Juicy Feedback — Aural and Visual Effects*, and besides the changes made in its **Create** event, the object is identical

For this game, the following five particle types will be created:

- Smoke used when the player's enemies die
- A health pickup particle
- The subparticles emitted while the health pickup particle is emitted
- A score pickup particle

- The subparticles emitted while the score pickup particle is emitted

Keeping in mind that `pt[n, 1]` represents the number of particles that will be burst, the following script creates and defines the parameters for these five particles:

```
/// Initializes variables for the particle systems.

// Creates a reference to the particle manager.
global.ps_manager = id;

// Creates a particle system.
ps = part_system_create();

// The number of particle types.
part_count = 5;

// Creates a puff of smoke that will be emitted when
enemies and/or the player are killed.
pt[0, 0] = part_type_create();
part_type_life(pt[0,0], 25, 65);
part_type_shape(pt[0,0], pt_shape_smoke);
part_type_color_mix(pt[0], c_ltgray, c_dkgray);
part_type_alpha2(pt[0,0], 1.0, 0.0);
part_type_size(pt[0,0], 0.8, 1.1, -0.005, 0);
part_type_speed(pt[0,0], 5, 10, -0.5, 0);
part_type_direction(pt[0,0], 0, 360, 0, 0);
part_type_orientation(pt[0,0], 0, 360, 0, 0, false);
pt[0, 1] = 40;

// Initializes second particle type.  These will be
released when collecting health pickups.
pt[1, 0] = part_type_create();
part_type_life(pt[1,0], 45, 45);
part_type_sprite(pt[1, 0], spr_pickup_health, true,
false, true);
part_type_blend(pt[1, 0], true);
part_type_alpha3(pt[1, 0], 1, 0.75, 0);
part_type_size(pt[1, 0], 1, 1, -0.01, 0);
part_type_speed(pt[1,0], 5, 10, 0, 4);
part_type_direction(pt[1,0], 0, 360, 12.5, 3.0);
pt[1, 1] = 3;

// Initializes second particle type.  These will be
released when collecting score pickups.
```

```
pt[2, 0] = part_type_create();
part_type_life(pt[2, 0], 45, 45);
part_type_sprite(pt[2, 0], spr_pickup_health, true,
false, true);
part_type_blend(pt[2, 0], true);
part_type_alpha2(pt[2, 0], 0.5, 0);
part_type_size(pt[2, 0], 0.75, 0.75, -0.01, 0);

// Will make the original particle emit one every five
steps causing a trail effect.
part_type_step(pt[1,0], 5, pt[2,0]);

// Initializes second particle type.  These will be
released when collecting score pickups.
pt[3, 0] = part_type_create();
part_type_life(pt[3,0], 45, 45);
part_type_sprite(pt[3, 0], spr_pickup_score, true,
false, true);
part_type_blend(pt[3, 0], true);
part_type_alpha2(pt[3, 0], 1, 0);
part_type_size(pt[3, 0], 1, 1, -0.01, 0);
part_type_speed(pt[3,0], 3, 7, 0, 4);
part_type_direction(pt[3,0], 0, 360, 6, 1);
pt[3, 1] = 3;

// Initializes second particle type.  These will be
released when collecting score pickups.
pt[4, 0] = part_type_create();
part_type_life(pt[4,0], 15, 15);
part_type_sprite(pt[4, 0], spr_pickup_score, true,
false, true);
part_type_blend(pt[4, 0], true);
part_type_alpha2(pt[4, 0], 0.5, 0);
part_type_size(pt[4, 0], 0.75, 0.75, -0.01, 0);

// Will make the original particle emit one every five
steps causing a trail effect.
part_type_step(pt[3,0], 5, pt[4,0]);
```

The important thing to note is that `part_count` is now set to `5` so that when the particle manager is destroyed, these five particles are properly cleaned up.

Once created, `obj_particle_manager` can be instantiated by `rm_level1`, with the following line of code in its **Creation Code**:

```
// Creates a new instance of the particle effect.
instance_create(0,0, obj_particle_manager);
```

Particle creation can appear to be a rather daunting task with a lot of trial and error, but fortunately, particle editors made by fans of GameMaker, such as Lithium Particle Designer by Sky Castle Games, do exist; however, knowing how to create and utilize particles from scratch is still an important skill.

Giving feedback with pickups

Instead of creating separate **Destroy** events for `obj_pickup_score` and `obj_pickup_health`, two new instanced variables, `snd_pitch` and `part_index`, will be added to their **Create** events and `scr_collect_pickup` will be updated. The following are the values set for the two previously mentioned variables in `obj_pick_health`:

```
// The pitch of the sound played when collected
snd_pitch = 1.0;

// The index of the particle played.
part_index = 1;
```

And here are those same variables in `obj_pickup_score`:

```
// The pitch of the sound played when collected
snd_pitch = 1.25;

// The index of the particle played.
part_index = 3;
```

The first variable, `snd_pitch`, will determine the pitch of the sound to be played, and `part_index` references the index of the particle to play when the pickup is destroyed. They are both used in `scr_pickup_score` with the following code:

```
// Destroys the orb while playing pick-up sound and
particle effects.
```

```
with (argument0)
{
    part_particles_create(global.ps_manager.ps, x, y,
global.ps_manager.pt[part_index, 0],
    global.ps_manager.pt[part_index, 1]);     var
snd_pickup = audio_play_sound(snd_pickup_collect, 0,
false);
    audio_sound_pitch(snd_pickup_collect, snd_pitch);

    instance_destroy();
}
```

Placed inside the final `with` statement of the script, the two previously created variables are used. First, `part_index` refers to the particles that will be played with the particle manager's two-dimensional array of particles. Then, `snd_pitch` is used in the built-in function `audio_sound_pitch`, which shifts the pitch of the sound created with `audio_play_sound`. This allows the same sound to be used but makes it slightly different.

These seemingly minor updates will allow the player to know they have collected something without them having to refer to the score or their healthbar to see that it has increased. At the same time, due to the trailed, swirling nature of the associated particles, the game has an added level of flare. The two particle effects are demonstrated in the following screenshot with the health pickup on the left and the score pickup on the right:



Providing feedback when Vlad is damaged

Another important moment of feedback that hasn't been implemented yet is when the player is damaged. Fortunately, the necessary resources have already been added. This code can be added to a variety of locations in the script, but it'll be added to `scr_damage_vlad` when the value of health is compared to `0`. The updated code is highlighted in the following code snippet:

```
// If the player's health is less than 0, the player
will die.
    if (health <= 0)
    {
        // Sets the state.
        state_id = state_death;
        entered_new_state = true;

        // Plays the death sound.
        audio_play_sound(snd_damage, 0, false);
// Plays the death particle.
        var burst_x, burst_y;
        burst_x = x;
        burst_y = y;
        with (global.ps_manager)
        {
            part_particles_create(ps, burst_x,
burst_y, pt[0,0], pt[0,1]);
        }
    }
    else
    {
        // Plays the damage sound at a higher pitch to
signal that it isn't as serious.
        var snd_hit = audio_play_sound(snd_damage, 0,
false);
        audio_sound_pitch(snd_hit, 1.1);
    }
```

So, in the previous code, if `health` is less than or equal to `0`, which means

the player has died, the damage sound is played. Then, the smoke particle at index `0` of the particle manager is emitted from the player's position. If the player is still alive, though, the sound is played with an increased pitch to signify that the damage done is not fatal.

Extending the enemy death sequence

When fighting enemies, sometimes it can make the player feel more accomplished if the enemies' deaths are more elaborate than just them disappearing suddenly. In this final segment, timelines will be used once again to give enemies a death sequence as opposed to them just disappearing.

Updating scr_collide_enemy and Step events for enemies

Before creating the timeline that will run when enemies perish, `scr_collide_enemy`—the script that is executed when Vlad collides with an enemy—must be updated with the following code; the highlighted portions are the changes made to the original script:

```
// If the player collides with an invisible object,
the function won't be continued.
if (argument0.timeline_index > -1)
{
    return 0;
}

// If the character is moving down and collides from
the top.
if (vspeed > 0 && bbox_bottom < argument0.bbox_bottom
&& bbox_top < argument0.bbox_top)
{
    // Makes the character jump.
    entered_new_state = true;
    grounded = true;
    state_id = state_jump;

    // Destroys the instance of the enemy and
increases the score.
    with (argument0)
    {
```

```
            // Increases the score.
            score += point_value;

            // Assigns the timeline and begins the death
sequence.
            timeline_index = tm_enemy_death;
            timeline_position = 0;
            timeline_running = true;
        }
    }
```

As mentioned earlier, timelines will be used to make the enemy die in a sequence instead of an instant, but this means that the enemy will remain on the screen longer and could possibly continue to collide with Vlad. To prevent this, the enemy's `timeline_index` object is checked before anything else is tested, returning `0` to end the function, if a timeline is assigned.

Then, within the `with` statement that occurs when the enemy dies, instead of using `instance_destroy`, a new timeline `tm_enemy_death` is used with `timeline_position` set to `0` and `timeline_running` set to `true`.

It's also important to make sure that the enemies' default **Step** events do not occur while they are dying, so the inclusion of the following code at the beginning of the **Step** events for both the `obj_enemy_garlic` and `obj_enemy_book` objects is very important:

```
if (argument0.timeline_index > -1)
{
    return 0;
}
```

Again, if the enemies' `timeline_index` object is assigned, their basic **Step** event will not be performed, assuming that the timeline has taken control.

Creating tm_enemy_death

This timeline resource, which is the final resource to be introduced in this chapter and book, will handle the death process for enemies. It will be

divided into four steps. Each segment will be relatively short and simple, but the idea of the delay and making events occur at different beats adds just a little bit of polish.

Step 0

First, the enemy will launch upward, flipping upside down. This will help give more reasons behind the extra bounce and launch that Vlad gets when jumping on enemies. At the same time, the damage sound will play, using a random pitch to show that an enemy has died. The value of this pitch is less than 0, so it sounds deeper and scarier than when Vlad is damaged. The following is the code for the first part of the timeline:

```
/// Part 1 of timeline for enemy death.

// Sets the vertical speed to ten.
vspeed = -10;

// Sets the gravity to 1.
gravity = 1;

// flips the character vertically.
image_yscale = -1;

// Stops animation on the enemy.
image_speed = 0;

// Plays a death sound.
var snd_death = audio_play_sound(snd_damage, 0,
false);

// Plays the sound at a random pitch.
audio_sound_pitch(snd_death, random_range(0.25,
0.75));
```

Step 5

At step 5, the enemy will turn semitransparent, signifying to the player that the enemy cannot be damaged anymore and that they cannot be damaged by the enemy, using the following code:

```
/// Part 2 of the enemy death timeline.

// Makes the enemy semitransparent.
image_alpha = 0.5;
```

Step 25

A little less than a second later, the smoke particle that is particle when Vlad dies will be emitted:

```
/// Part 3 of the enemy death timeline.

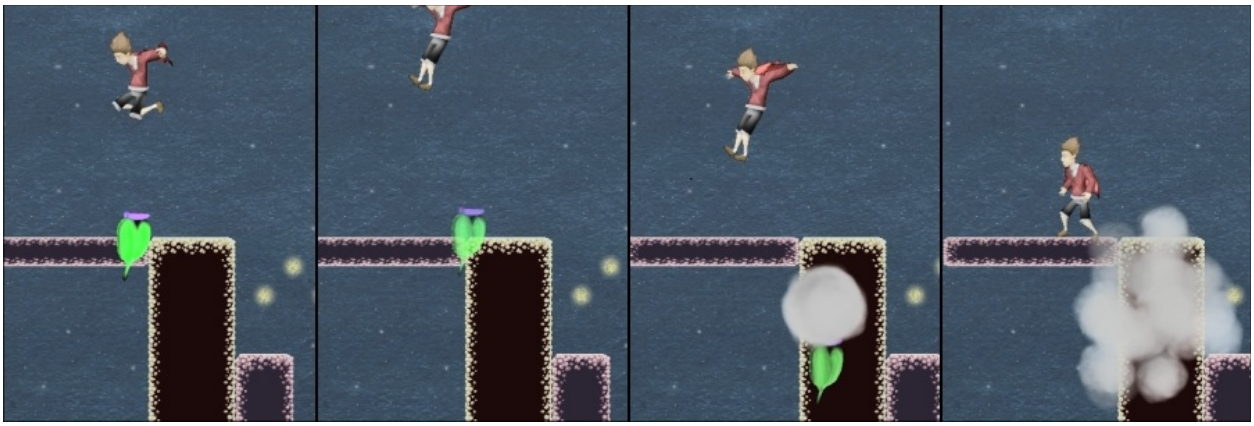
// Death particle is emitted.
var burst_x, burst_y;
burst_x = x;
burst_y = y;
with (global.ps_manager)
{
    part_particles_create(ps, burst_x, burst_y,
pt[0,0] , pt[0,1]);
}
```

Step 30

At the final moment of the timeline, the enemy instance is destroyed with the following code:

```
/// Part 4 of the enemy death timeline.
// Destroys the enemy.
instance_destroy();
```

A lot of the previous code should be familiar as it uses variables and functions used in earlier parts of this chapter and text. This timeline should make the enemy's death appear similar to the sequence displayed in the following screenshot:



It may not be the most elaborate or satisfying death sequence, but the utilization of timelines is definitely a good place to start.

Summary

In this chapter, a goal in the form of a mysterious door was added to the platformer game for players to strive towards. Timelines were introduced and then used so that when players reach this door, a series of scripts will be executed instead of the character instantaneously going to the next room or level.

Then, feedback concepts were reviewed, such as playing audio and shifting its pitch so that the same sounds can be recycled, ultimately saving resources. The particle effect manager was also reviewed and re-used. Finally, a second timeline was used when enemies perish.

There are still tons of ideas that could be gone over before making this platformer a shippable product, but the hope is that enough has been taught that the tools available can now be used to create any kind of game you desire and bring it to a fun, polished, and possibly shippable state.

In closing...

The goal of this book was never to create a final, shippable game, but instead, to help teach the powerful scripting tools that GameMaker provides to create games. It goes without saying that GameMaker and GML are not the end-all-be-all techniques for making games. There are dozens, if not hundreds, of different engines, frameworks, and languages out there, but everyone has to start somewhere, and GameMaker is a great place to do so.

Hopefully, this book stands as a strong learning point for all kinds of readers: the hobbyist making their first game; a designer trying to create convincing, inspiring prototypes; or a programmer who is simply exploring a new programming language. Regardless of the tool used to create a game, though, the design is what will really makes it special. Is this game innovative? How is this game going to change the genre? Is it going to invent a new genre? Lately, games are often accused of being clones or lacking true innovation, so it's extremely important to try and introduce originality and uniqueness to any game created. There are tons of resources out there for creating games, but there are few—if any—tools out there to suddenly bring innovation or originality to a game. That begins with the game makers.

Index

A

- absolute value (abs) / scr_swap_pieces
- Alarm 0 event
    - setting / Setting the Alarm 0 event of obj_puzzle_piece
- Alarm 1 event
    - using, in obj_grid_manager / Alarm 1
- Alarm 2 event
    - using, in obj_grid_manager / Alarm 2
- alarms
    - about / Introducing alarms
    - adding, for puzzle game / Arming the puzzle game's alarms
    - setting up, for puzzle game / Setting up the first alarm, Setting the alarms
    - setting, to delay pieces shifting / Are pieces shifting?
    - used, for determining pieces position / Determining where pieces should move
    - Alarm 0 event, setting / Setting the Alarm 0 event of obj_puzzle_piece
    - scr_swap_pieces, updating / Updating scr_swap_pieces
    - scr_check_board, updating / Updating scr_check_board
    - scr_reorganize_board function, updating / Updating scr_reorganize_board
- alpha tolerance
    - about / Sprites – spr_button

- arguments
    - about / Functions and accessing script resources, Arguments
- arrays
    - about / Arrays
    - one-dimensional array / Arrays
    - two-dimensional array / Two-dimensional arrays
- aspect ratios / Setting the view
- audio effects
    - adding, to puzzle game / Hiding the pin drop – adding audio
    - audio resources, creating / Creating sound resources
    - audio resources, gathering / Gathering audio resources
    - audio functions / Introducing audio functions
    - playing / Playing puzzling sounds
    - sounds, swapping in scr_swap_pieces / Swapping sounds in scr_swap_pieces
    - pop sound, playing in scr_reorganize_board / Playing the pop sound in scr_reorganize_board
    - music, fading in with obj_grid_manager / Fading in the music with obj_grid_manager
- audio functions
    - about / Introducing audio functions
- audio resources
    - creating / Creating sound resources
    - gathering / Gathering audio resources

B

- background index

    - setting / Setting a background index with bg_index

- background resource
    - about / Setting the environment – backgrounds

- background resources
    - creating / Creating background resources
    - preparing / Preparing the game for background resources
    - atmosphere, building / Building an atmosphere with bg_main
    - utilizing / Utilizing the background

- backgrounds
    - setting / Setting the environment – backgrounds
    - background variables, creating / Introducing background variables
    - background resources, creating / Creating background resources
    - background resources, preparing / Preparing the game for background resources
    - background index, setting / Setting a background index with bg_index
    - scr_inverse_lerp, scripting / Scripting scr_inverse_lerp
    - moving, within End Step event / Moving the background in the End Step event of obj_camera

- background variables
    - creating / Introducing background variables

- bgm_normal music track / Gathering audio resources
- boolean comparison
    - about / Expression symbols

- break statement / break, continue, and return
- built-in variables

    - about / Built-in variables

- buttons
    - creating / Pushing your buttons
    - project, creating / Creating the project
    - resources, gathering / Gathering resources
    - Draw GUI event, adding / The Draw GUI event
    - creating, scripts used / Creating buttons using scripts
    - room creation code, scripting / Scripting room creation code
    - scr_random_position, creating / Creating scr_random_position
    - exporting / Exporting and importing the button
    - importing / Exporting and importing the button

C

- camera object
    - creating / Preparing the game for obj_camera
    - Create event, scripting / The Create event
    - End Step event, scripting / The End Step event
    - adding, with creation code / Adding the camera with Creation Code

- collision
    - creating / Collision – a crash course
    - masks, creating / Creating masks for collision
    - placement function, using / Testing placement
    - movement function / Movement functions
    - Collision events / Working with Collision events
    - handling, for garlic enemy / Reacting upon collision with scr_collide_enemy

- Collision events
    - adding / Working with Collision events
    - Create event, updating / Updating the Create event of obj_vlad

    - creating, of obj_vlad / The Collision events of obj_vlad
    - scr_test_collision script, creating / Creating the script resource scr_test_collision
    - updating, with Step event / Updating the Step event

- color
    - working with / Working with frames and color

- combo
    - about / Scoring and combos
    - adding / Scoring and combos

- comment block
    - about / Commenting

- comments
    - about / Commenting

- compile errors
    - about / Errors

- conditional statements
    - about / Conditional statements
    - if statement / if, else, and switch
    - if-else statement / if, else, and switch
    - switch statement / if, else, and switch
    - repeat statement / repeat, while, do, and for
    - while statement / repeat, while, do, and for
    - do statement / repeat, while, do, and for
    - for statement / repeat, while, do, and for
    - break / break, continue, and return
    - continue / break, continue, and return
    - return / break, continue, and return

- constants
    - about / Variable scope
    - creating / Creating custom constants
- continue statement / break, continue, and return
- Create event
    - adding / The Create event
- creation code
    - camera object, adding with / Adding the camera with Creation Code

D

- damage sound
    - implementing, for platformer game / Providing feedback when Vlad is damaged
- death state, hazards
    - creating / Establishing a death state
    - User Defined 3 event, setting up / Setting up two new events – User Defined 3 and Alarm 1
    - Alarm 1 event, setting up / Setting up two new events – User Defined 3 and Alarm 1
    - User Defined 3, setting up / Setting up two new events – User Defined 3 and Alarm 1
- degrees / Rotating
- distributions, particle emitters / Shaping and distributing within emitters
- div function
    - using / Utilizing div and mod

- do statement
  - about / repeat, while, do, and for
  - differentiating, with while statement / repeat, while, do, and for
- Draw event
  - about / Drawing and graphical user interface
  - draw functions / Understanding draw functions
  - drawing order, establishing / Establishing the drawing order
- draw functions
  - about / Understanding draw functions
  - rectangle, drawing / Drawing rectangles
  - color, setting for rectangle / Setting color and alpha
  - alpha, setting for rectangle / Setting color and alpha
  - circles, drawing / Drawing circles and ellipses
  - ellipses, drawing / Drawing circles and ellipses
  - circle precision, setting / Setting a circle's precision
  - points, drawing / Drawing points, lines, and arrows
  - lines, drawing / Drawing points, lines, and arrows
  - arrows, drawing / Drawing points, lines, and arrows
  - text, drawing / Drawing text
  - text font, setting / Setting font and text alignment
  - text alignment, setting / Setting font and text alignment
  - sprites, drawing / Drawing sprites
  - sprites, tiling / Tiling sprites
- Draw GUI event
  - adding / The Draw GUI event
  - scr_create_button script, using / Scripts – scr_create_button
  - scr_random_position script, using / scr_random_position
  - about / Drawing and graphical user interface
- drawing order
  - establishing, in Draw event / Establishing the drawing order

E

- enemies, platformer game
    - creating / Fighting the player with enemies
    - flying book enemy / Enemy 2 – the flying book
- enemy death sequence
    - scr_collide_enemy, updating / Updating scr_collide_enemy and Step events for enemies
    - Step events, updating / Updating scr_collide_enemy and Step events for enemies
    - timeline resource, creating / Creating tm_enemy_death
- errors
    - about / Errors
    - compile error / Errors
    - runtime error / Errors
- event_perform function
    - using / event_perform
- expressions
    - about / Expressions
    - symbols / Expression symbols
    - with mathematical operations / Expression symbols
- expression symbols
    - about / Expression symbols

F

- falling offscreen, hazards
    - implementing / Falling off the screen into the abyss
- feedback
    - reviewing / Reviewing polish, feedback, and juiciness
    - sound resources, creating / Gathering resources to play sounds
    - music, playing with scr_play_music / Playing music with scr_play_music
    - particle manager, reviewing / Reviewing the obj_particle_manager object
    - giving, with pickups / Giving feedback with pickups
- finite state machine (FSM)
    - about / Introducing finite state machines
- flying book enemy
    - about / Enemy 2 – the flying book
    - resources, gathering / Gathering resources to create the flying book
    - sprite resource, creating / The sprite resource – spr_enemy_book
    - object resource, creating / The object resource – obj_enemy_book
    - Create event, initializing of object resource / Initializing the Create event of obj_enemy_book
    - Step event, scripting / Anthropomorphizing the book in the Step event
- fnt_main
    - about / Gathering resources for creating the main menu
    - creating / Creating fonts – fnt_title and fnt_main
    - setting / Creating the fonts
- fnt_title

    - about / Gathering resources for creating the main menu
    - creating / Creating fonts – fnt_title and fnt_main
    - setting / Creating the fonts

- for statement
    - about / repeat, while, do, and for

- frames
    - working with / Working with frames and color

- functions
    - about / Functions and accessing script resources
    - used, for accessing script resource / Functions and accessing script resources
    - arguments / Functions and accessing script resources, Arguments

G

- garlic enemy
    - about / Enemy 1 – mutant garlic
    - resources, gathering / Gathering resources for garlic
    - sprite resource, creating / Sprite resource – spr_enemy_garlic
    - object resource, creating / Object resource – obj_enemy_garlic
    - Create event, scripting for object resource / Scripting the Create event of obj_enemy_garlic
    - moving, with Step event / Making garlic move with the Step event
    - collision, handling / Reacting upon collision with scr_collide_enemy
    - Collision event, adding to obj_vlad / Colliding with the player – the Collision event of obj_vlad

- Global left release event

    - adding / The Global left release event
- global variable scope
    - about / Variable scope
- GML scripts
    - creating / Creating GML scripts
    - creating, within event / Creating GML scripts within an event
    - creating, as resource / Creating scripts as resources
    - creating, as rooms creation code / Scripting a room's creation code
    - program / Programs
    - variables / Variables
    - functions / Functions and accessing script resources
    - expressions / Expressions
    - conditional statements / Conditional statements
    - arrays / Arrays
    - comments / Commenting
    - errors / Errors
- goal
    - setting, for platformer game / GOAL!
    - resources, gathering / Gathering resources for creating the goal
    - sprite resource, creating / Sprite resource – spr_goal_door
    - object resource, creating / Object resource – obj_goal_door
- grid
    - puzzle game pieces, aligning to / Aligning the pieces of the puzzle game to a grid

H

- hazards

    - creating / Dying from hazards
    - death state, creating / Establishing a death state
    - resources, gathering / Gathering resources for hazards
    - object resources, creating / Creating the sprite and object resources
    - sprite resources, creating / Creating the sprite and object resources
    - scr_damage_vlad, scripting / Scripting scr_damage_vlad
    - falling offscreen, implementing / Falling off the screen into the abyss

- idle state
    - about / Introducing finite state machines
- if-else statement
    - about / if, else, and switch
- if statement
    - about / if, else, and switch
- index
    - about / Arrays
- infinite loop
    - about / repeat, while, do, and for
- instance variable scope
    - about / Variable scope
- is_shifting_pieces function

    - applying / Applying is_shifting_pieces

J

- jump sounds, platformer game
    - creating / Jumping with sound – snd_jump
- jump state
    - about / Introducing finite state machines

K

- keyboard events
    - Keyboard event / Introducing keyboard functions, variables, and events
    - Key Press event / Introducing keyboard functions, variables, and events
    - Key Release event / Introducing keyboard functions, variables, and events
    - about / Introducing keyboard functions, variables, and events
- keyboard functions
    - about / Introducing keyboard functions, variables, and events
- keyboard input
    - integrating, with puzzle game / Integrating keyboard input
    - keyboard functions / Introducing keyboard functions, variables, and events
    - keyboard variables / Introducing keyboard functions, variables, and events
    - keyboard events / Introducing keyboard functions, variables, and events
    - Keyboard events, updating / Integrating the Keyboard event

updates
    - Key Press event, utilizing / Utilizing the press <any key> event
- keyboard variables
    - about / Introducing keyboard functions, variables, and events

L

- Left Button event
    - adding / The Left Button event
- Left Pressed event
    - adding / The Left Pressed event
- local variable scope
    - about / Variable scope
- loop
    - about / repeat, while, do, and for

M

- main menu
    - creating / Gathering resources for creating the main menu
    - obj_main_menu / Gathering resources for creating the main menu
    - rm_main_menu / Gathering resources for creating the main menu
    - scr_on_menu_button_pressed / Gathering resources for creating the main menu
    - fnt_main / Gathering resources for creating the main menu
    - fnt_title / Gathering resources for creating the main menu

    - fnt_title, creating / Creating fonts – fnt_title and fnt_main
    - fnt_main, creating / Creating fonts – fnt_title and fnt_main
    - fnt_main, setting / Creating the fonts
    - fnt_title, setting / Creating the fonts

- masks
    - creating, for collision / Creating masks for collision
- mod function
    - using / Utilizing div and mod
- moments / Introducing timelines
- Mouse Enter event
    - adding / The Mouse Enter event
- mouse events
    - implementing, for obj_grid_block / Implementing the Mouse events for obj_grid_block
- mouse input
    - puzzle game pieces, swapping / Swapping pieces
    - scr_check_board script, updating / Updating scr_check_board
- Mouse Leave event
    - adding / The Mouse Leave event
- movement function
    - about / Movement functions
- move_bounce_all(use_precision) function / Movement functions
- move_bounce_solid(use_precision) function / Movement functions
- move_contact_all(direction, max_distance) function / Movement functions

- move_contact_solid(direction, max_distance) function / Movement functions
- move_outside_all(direction, max_distance) function / Movement functions
- move_outside_solid(direction, max_distance) function / Movement functions
- move_random(horizontal_snap, vertical_snap) function / Movement functions
- move_snap(horizontal_snap, vertical_snap) function / Movement functions
- move_towards_point(x, y, speed) function / Movement functions
- move_wrap(wrap_horizontal, wrap_vertical, margin) function / Movement functions
- moving platform
    - with path / Moving platforms with paths
    - path resources, creating / Creating path resources
    - path_start function, utilizing / Utilizing the path_start function
    - resources, gathering for path creation / Gathering resources for the path creation
    - spr_moving_platform sprite / The spr_moving_platform sprite
    - obj_moving_platform / obj_moving_platform
    - pth_horizontal / pth_horizontal and pth_vertical_ellipse
    - pth_vertical_ellipse / pth_horizontal and pth_vertical_ellipse
    - integrating, into room / Integrating the moving platforms
    - instances, creating of obj_moving_platform / Creating instances of obj_moving_platform
    - interacting, with obj_moving_platform / Interacting with obj_moving_platform
    - current_platform, creating of obj_vlad / Creating a new variable – current_platform
    - User defined event, updating of obj_vlad / Updating the User Defined 2 event
    - current_platform, updating of obj_vlad / scr_test_collision and current_platform
    - scr_test_collision, updating of obj_vlad / scr_test_collision and current_platform

    - Step End event, using / Using the Step End event
    - path, drawing / Drawing a path
- music, feedback
    - playing, with scr_play_music / Playing music with scr_play_music

N

- new project
    - creating, for buttons / Creating the project
- nonuniform scaling / Scaling

O

- object resource, flying book enemy
    - creating / The object resource – obj_enemy_book
    - Create event, initializing / Initializing the Create event of obj_enemy_book
- object resource, garlic enemy
    - creating / Object resource – obj_enemy_garlic
    - Create event, scripting / Scripting the Create event of obj_enemy_garlic
- object resource, goal
    - creating / Object resource – obj_goal_door
    - updating / Gathering resources for integrating the timeline
    - font resource, creating / Font resource – fnt_large_text
    - Create event, updating / Creating and updating events for obj_goal_door
    - Animation End event, using / Using the Animation End event

    - congratulatory text, drawing / Drawing congratulatory text with the Draw GUI event
    - instance, deactivating with scr_deactive_instances / Deactivating objects with scr_deactive_instances and the Draw event
    - instance, deactivating with Draw event / Deactivating objects with scr_deactive_instances and the Draw event
    - instance, freezing / Freezing instances with scr_deactivate_instances
    - deactivated instance, drawing / Drawing deactivated instances

- object resource, pickups
    - initializing / Pickup object resources

- object resource, platformer game
    - creating / Creating a new object resource – obj_vlad
    - User defined events, utilizing / Utilizing the User defined events

- object resources, hazards
    - creating / Creating the sprite and object resources

- obj_basic_platform
    - about / Objects – obj_basic_platform and obj_solid_platform

- obj_button object
    - about / Objects – obj_button
    - Create event, adding / The Create event
    - Left Button event, adding / The Left Button event
    - Left Pressed event, adding / The Left Pressed event
    - Mouse Enter event, adding / The Mouse Enter event
    - Mouse Leave event, adding / The Mouse Leave event
    - Global left release event, adding / The Global left release event

- obj_grid_block

    - mouse events, implementing / Implementing the Mouse events for obj_grid_block
    - Mouse Enter event, implementing / Entering the grid with the mouse
    - Left Pressed event, implementing / Pressing the left mouse button
    - Global Left Released event, implementing / Releasing the left mouse button globally
    - Mouse leave event, implementing / The Mouse Leave event

- obj_grid_manager

    - modifying / Changing obj_grid_manager

- obj_main_menu

    - about / Creating obj_main_menu
    - scripting / Scripting obj_main_menu
    - Create event / The Create event
    - div function, using / Utilizing div and mod
    - mod function, using / Utilizing div and mod
    - Draw event, scripting / Drawing the menu
    - Global Left Button event, scripting / The Global Left Button event
    - press <any key> event, scripting / The press <any key> event
    - Global Left Release event, scripting / The Global Left Release event
    - release <any key> event, scripting / The release <any key> event
    - event_perform function, using / event_perform
    - scr_on_menu_button_pressed, scripting / scr_on_menu_button_pressed

- obj_particle_manager object

    - creating / obj_particle_manager
    - Create event / The Create event
    - Destroy event / The Destroy event

    - placing / Placing obj_particle_manager
- obj_solid_platform
    - about / Objects – obj_basic_platform and obj_solid_platform
    - solidifying / Solidifying objects
- one-dimensional array
    - about / Arrays
- origin, sprite variables / Positioning and origin

P

- parallax
    - about / Moving the background in the End Step event of obj_camera
- particle-specific functions
    - using / Particle-specific functions
- particle effects
    - about / Visualizing effects with particles
    - creating, with particle systems / Visualizing effects with particles
    - creating, with particle emitters / Visualizing effects with particles
    - creating, with particle types / Visualizing effects with particles
    - particle-specific functions / Particle-specific functions
    - integrating, into puzzle game / Integrating particles
    - integrating, obj_particle_manager object used / obj_particle_manager
    - bursts, creating within scr_reorganize_board / Creating bursts within scr_reorganize_board
- particle emitters

    - about / Visualizing effects with particles
    - using / Emitting particles
    - distributions / Shaping and distributing within emitters
    - shapes / Shaping and distributing within emitters

- particle manager, feedback
    - reviewing / Reviewing the obj_particle_manager object

- particle systems
    - about / Visualizing effects with particles
    - managing / Managing a particle system

- particle types
    - about / Visualizing effects with particles
    - creating / Creating different particle types
    - shapes / Shaping particle types

- part_emitter_region
    - ps_distr_linear / Shaping and distributing within emitters
    - ps_distr_gaussian / Shaping and distributing within emitters
    - ps_distr_invgaussian / Shaping and distributing within emitters

- path
    - creating, for moving platform / Moving platforms with paths
    - drawing, for moving platform / Drawing a path

- path resources
    - creating, for moving platform / Creating path resources
    - pth_vertical_ellipse / pth_horizontal and pth_vertical_ellipse
    - pth_horizontal / pth_horizontal and pth_vertical_ellipse

- path_start function

    - utilizing / Utilizing the path_start function

- pickups
    - about / Working with pickups
    - resources, gathering / Gathering resources to create pickups
    - sprite resource, initializing / Initializing sprite resources for pickups
    - object resource, initializing / Pickup object resources
    - script resource, initializing / Script resource – scr_collect_pickup
    - Collision event, adding to obj_vlad / Colliding with obj_vlad
    - used, for giving feedback / Giving feedback with pickups

- placement function
    - using / Testing placement

- place_empty(x, y) function / Testing placement
- place_free(x, y) function / Testing placement
- place_meeting(x, y, obj) function / Testing placement
- place_snapped(horizontal_snap, vertical_snap) function / Testing placement
- platform
    - resources, gathering / Gathering resources to build platforms

- platformer game
    - resources, gathering / Gathering resources for the platformer
    - Vlad sprites, creating / Establishing Lil' Vlad's sprites
    - jump sounds, creating / Jumping with sound – snd_jump
    - object resource, creating / Creating a new object resource – obj_vlad
    - room resource, creating / Placing Vlad in a room – rm_level1
    - Vlad state constants, defining / Defining Vlad's state constants
    - collision, creating / Collision – a crash course
    - room, populating with platform / Populating the room
    - moving platform, with path / Moving platforms with paths

    - Vlad movement, handling / Preventing Vlad from leaving
    - features, designing ahead / Knowing the design ahead of time (when possible)
    - enemies, creating / Fighting the player with enemies
    - goal, setting / GOAL!

- position, sprite variables / Positioning and origin
- program
    - about / Programs
    - snake case / snake_case

- puzzle game
    - setting up / Setting up the puzzle game
    - spr_grid sprite / Sprites – spr_grid
    - spr_pieces sprite / spr_pieces
    - obj_grid_manager object / Objects – obj_grid_manager
    - obj_grid_block object / obj_grid_block and obj_puzzle_piece
    - obj_puzzle_piece object / obj_grid_block and obj_puzzle_piece
    - rm_gameplay room / Room – rm_gameplay
    - scripts, creating / Scripts
    - randomization, implementing / Randomizing the board
    - executing / Running it all together
    - keyboard input, integrating with / Integrating keyboard input
    - alarms, adding / Arming the puzzle game's alarms
    - alarms, setting up / Setting up the first alarm
    - audio effects, adding to / Hiding the pin drop – adding audio
    - particle effects, integrating / Integrating particles
    - timer, integrating to / Integrating score and time, Timing and game over

- puzzle game pieces
    - aligning, to grid / Aligning the pieces of the puzzle game to a grid
    - obj_grid_manager event, creating / The Create event of obj_grid_manager

    - scr_create_in_grid, coding / scr_create_in_grid
    - checking / Checking pieces
    - checking, with scr_get_puzzle_piece script / scr_get_puzzle_piece
    - checking, with scr_check_adjacent script / scr_check_adjacent
    - checking, with scr_check_board script / scr_check_board
    - swapping / Swapping pieces
    - obj_puzzle_piece, creatiing / obj_puzzle_piece
    - scr_swap_pieces, creating / scr_swap_pieces
    - updating, with scr_reorganize_board script / Updating organization with scr_reorganize_board

R

- radians / Rotating
- random functions
    - about / Random functions
- randomization
    - about / Understanding and utilizing randomization
    - random functions / Random functions
    - implementing, in puzzle game / Randomizing the board
- recursive process
    - about / Checking pieces
- repeat statement
    - about / repeat, while, do, and for
- resources
    - gathering, for buttons / Gathering resources
    - spr_button sprite / Sprites – spr_button
    - obj_button object / Objects – obj_button

    - rm_main room / Room – rm_main

- resources, moving platform
    - gathering, for path creation / Gathering resources for the path creation

- resources, platform
    - gathering / Gathering resources to build platforms
    - spr_basic_platform / Sprites – spr_basic_platform and spr_solid_platform
    - spr_solid_platform / Sprites – spr_basic_platform and spr_solid_platform
    - obj_basic_platform / Objects – obj_basic_platform and obj_solid_platform
    - obj_solid_platform / Objects – obj_basic_platform and obj_solid_platform
    - obj_solid_platform, solidifying / Solidifying objects

- resources, platformer game
    - gathering / Gathering resources for the platformer

- return statement / break, continue, and return
- rm_main_menu
    - about / Building a new room – rm_main_menu

- room
    - expanding, with views / Expanding the room – views

- room, platformer game
    - populating, with platform / Populating the room

- room resource, platformer game

    - creating / Placing Vlad in a room – rm_level1
- rotation
    - about / Rotating
- runtime errors
    - about / Errors

S

- scaling
    - about / Scaling
    - nonuniform scaling / Scaling
    - uniform scaling / Scaling
- scores
    - integrating, to puzzle game / Integrating score and time
    - adding / Scoring and combos
    - variables, initiating in Create event / The Create event of obj_grid_manager
    - points, earning in scr_reorganize_board / Earning points in scr_reorganize_board
    - displaying, with Draw event in obj_grid_manager / Drawing UI in obj_grid_manager
- script resource, pickups
    - initializing / Script resource – scr_collect_pickup
- scr_check_adjacent script / Scripts
    - puzzle game pieces, checking with / scr_check_adjacent
- scr_check_board script / Scripts

    - puzzle game pieces, checking with / scr_check_board
    - updating / Updating scr_check_board

- scr_create_in_grid script / Scripts
- scr_get_puzzle_piece script / Scripts

    - puzzle game pieces, checking with / scr_get_puzzle_piece

- scr_inverse_lerp

    - scripting / Scripting scr_inverse_lerp

- scr_on_menu_button_pressed

    - about / Gathering resources for creating the main menu
    - scripting / scr_on_menu_button_pressed

- scr_reorganize_board function

    - updating / Updating scr_reorganize_board
    - bursts, creating within / Creating bursts within scr_reorganize_board

- scr_swap_pieces function

    - updating / Updating scr_swap_pieces

- shapes, particle emitters / Shaping and distributing within emitters
- shapes, particle types / Shaping particle types
- snake case

    - about / snake_case

- snd_incorrect_swap effect / Gathering audio resources
- snd_pop effect / Gathering audio resources
- snd_swap effect / Gathering audio resources
- sound resources, feedback

    - creating / Gathering resources to play sounds

- sprite functions
  - about / Understanding sprite variables and functions
- sprite resource, flying book enemy
  - creating / The sprite resource – spr_enemy_book
- sprite resource, garlic enemy
  - creating / Sprite resource – spr_enemy_garlic
- sprite resource, goal
  - creating / Sprite resource – spr_goal_door
- sprite resource, pickups
  - initializing / Initializing sprite resources for pickups
- sprite resources, hazards
  - creating / Creating the sprite and object resources
- sprites
  - drawing, with draw functions / Drawing sprites
  - tiling, with draw functions / Tiling sprites
- sprite variables
  - about / Understanding sprite variables and functions
  - position / Positioning and origin
  - origin / Positioning and origin
  - layering, with depth / Layering with depth
  - rotating / Rotating
  - scaling / Scaling
  - frames, working with / Working with frames and color
  - color, working with / Working with frames and color

- spr_basic_platform
  - about / Sprites – spr_basic_platform and spr_solid_platform
- spr_button sprite
  - about / Sprites – spr_button
  - subimages / Sprites – spr_button
  - origin / Sprites – spr_button
  - bounds / Sprites – spr_button
- spr_solid_platform
  - about / Sprites – spr_basic_platform and spr_solid_platform
- state
  - about / Introducing finite state machines
- switch statement
  - about / if, else, and switch

T

- tile resources
  - building / Building resources for tiles
  - utilizing / Tiling with bg_tiles_main
- tiles
  - about / Introducing tiles
  - creating / Creating tiles
  - defining, with parameters / Creating tiles
  - tile resources, building / Building resources for tiles
  - applying / Applying tiles
  - tile_add function, utilizing / Utilizing tile_add and other tile functions

    - placing, with scr_define_tiles scripts / Placing tiles with scripts – scr_define_tiles
    - scr_define_tiles scripts, using / Using scr_define_tiles
    - drawing, in obj_moving_platform / Drawing tiles in obj_moving_platform

- tile_add function

    - utilizing / Utilizing tile_add and other tile functions
    - arguments / Utilizing tile_add and other tile functions

- Timeline Properties window

    - Add button / Introducing timelines
    - Change button / Introducing timelines
    - Delete button / Introducing timelines
    - Clear button / Introducing timelines
    - Shift button / Introducing timelines
    - Duplicate button / Introducing timelines
    - Spread button / Introducing timelines
    - Merge button / Introducing timelines

- timeline resource, enemy death sequence

    - step 0 / Step 0
    - step 5 / Step 5
    - step 25 / Step 25
    - step 30 / Step 30

- timelines

    - about / Introducing timelines
    - using / Using timelines
    - creating / Creating and applying the timeline
    - applying / Creating and applying the timeline
    - moments, step 0 / Step 0
    - moments, step 15 / Step 15
    - moments, step 90 / Step 90

    - tm_finish_level, applying / Applying tm_finish_level

- timer
    - integrating, to puzzle game / Integrating score and time, Timing and game over
    - variables, initiating in Create event of obj_grid_manager / Adding new variables to the Create event of obj_grid_manager
    - Alarm 1 event, using in obj_grid_manager / Alarm 1
    - Alarm 2 event, using in obj_grid_manager / Alarm 2
    - drawing, with Draw event / Drawing the timer
- two-dimensional array
    - about / Two-dimensional arrays

U

- uniform scaling / Scaling

V

- variable prefixes
    - about / Variable prefixes
- variables
    - about / Variables
    - prefixes / Variable prefixes
    - variable scope / Variable scope
- variable scope
    - about / Variable scope
    - instance / Variable scope
    - local / Variable scope
    - global / Variable scope

    - constants / Variable scope
    - built-in variables / Built-in variables
    - custom constants, creating / Creating custom constants

- view parameters
    - adjusting / Adjusting view parameters, Preparing the game for obj_camera

- views
    - room, expanding with / Expanding the room – views
    - setting / Setting the view

- Vlad events
    - scripting / Starting Vlad's events and scripts – walking
    - Create event, scripting / The Create event
    - Step event, scripting / The Step event
    - User Defined 0 event, scripting for idle state / Standing still – the User Defined 0 event
    - User Defined 1 event, scripting for walk state / Walk this way – the User Defined 1 event
    - new variables, adding for jump state / Adding new variables for jumping
    - Step event, updating for jump state / Using up to jump – the Step event update
    - User Defined 2 event, scripting for fall state / Falling state – the User Defined 2 event
    - Animation End event, scripting / Looping the jump – the Animation End event

- Vlad health
    - tracking, with Draw events / Tracking health with Draw and Draw GUI
    - tracking, with Draw GUI events / Tracking health with Draw and Draw GUI

    - UI, displaying / Displaying UI with the Draw and Draw GUI events

- Vlad health UI
    - displaying, with Draw event / Displaying UI with the Draw and Draw GUI events, Displaying health with the Draw event
    - displaying, with Draw GUI event / Displaying UI with the Draw and Draw GUI events
    - new font, creating / Creating a new font – fnt_score_text
    - Draw GUI event, setting up / Setting up the Draw GUI event
    - Create event, updating / Updating the Create event
    - Alarm 0 event, scripting / Arming the Alarm 0 event in obj_vlad

- Vlad movement
    - handling / Preventing Vlad from leaving
    - global.room_left, defining / Defining global.room_left and global.room_right
    - global.room_right, defining / Defining global.room_left and global.room_right
    - End Step event, updating / Updating the End Step event

- Vlad sprites, platformer game
    - spr_vlad_idle sprite / The standing sprite – spr_vlad_idle
    - spr_vlad_walk sprite / The walking sprite – spr_vlad_walk
    - spr_vlad_jump sprite / The jumping sprite – spr_vlad_jump

- Vlad state
    - idle state / Introducing finite state machines
    - walk state / Introducing finite state machines
    - jump state / Introducing finite state machines

- Vlad state constants
    - defining / Defining Vlad's state constants

W

- walk state
  - about / Introducing finite state machines
- while statement
  - about / repeat, while, do, and for
  - differentiating, with do statement / repeat, while, do, and for